W3School Go语言 教程

wizardforcel

Published with GitBook



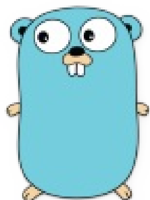
目錄

W3School Go语言 教程

作者：W3School

来源：Go语言 教程

Go 语言简介



Go 是一个开源的编程语言，它能让构造简单、可靠且高效的软件变得容易。

Go是从2007年末由Robert Griesemer, Rob Pike, Ken Thompson主持开发，后来还加入了Ian Lance Taylor, Russ Cox等人，并最终于2009年11月开源，在2012年早些时候发布了Go 1稳定版本。现在Go的开发已经是完全开放的，并且拥有一个活跃的社区。

Go 语言特色

- 简洁、快速、安全
- 并行、有趣、开源
- 内存管理、v数组安全、编译迅速

Go 语言用途

Go 语言被设计成一门应用于搭载 Web 服务器，存储集群或类似用途的巨型中央服务器的系统编程语言。

对于高性能分布式系统领域而言，Go 语言无疑比大多数其它语言有着更高的开发效率。它提供了海量并行的支持，这对于游戏服务端的开发而言是再好不过了。

第一个 Go 程序

接下来我们来编写第一个 Go 程序 hello.go（Go 语言源文件的扩展是 .go），代码如下：

实例

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World!")
}
```

运行实例 »

执行以上代码输出

```
$ go run hello.go Hello,  World!
```

Go 语言环境安装

Go 语言支持以下系统：

- Linux
- FreeBSD
- Mac OS X（也称为 Darwin）
- Window

安装包下载地址为：https://golang.org/dl/。

各个系统对应的包名：

操作系统	包名
Windows	go1.4.windows-amd64.msi
Linux	go1.4.linux-amd64.tar.gz
Mac	go1.4.darwin-amd64-osx10.8.pkg
FreeBSD	go1.4.freebsd-amd64.tar.gz

File name	Kind	OS	Arch	SHA1 Checksum
go1.4.2.src.tar.gz	**Source**			**460caac03379f746c473814a65223397e9c9a2f6**
go1.4.2.darwin-386-osx10.6.tar.gz	Archive	OS X 10.6+	32-bit	fb3e6b30f4e1b1be47bbb98d79dd53da8dec24ec
go1.4.2.darwin-386-osx10.8.tar.gz	Archive	OS X 10.8+	32-bit	65f5610fdb38febd869aeffbd426c83b650bb408
go1.4.2.darwin-386-osx10.6.pkg	Installer	OS X 10.6+	32-bit	3ed569ce33616d5d36f963e5d7cefb55727c8621
go1.4.2.darwin-386-osx10.8.pkg	Installer	OS X 10.8+	32-bit	7f3fb2438fa0212febef13749d8d144934bb1c80
go1.4.2.darwin-amd64-osx10.6.tar.gz	Archive	OS X 10.6+	64-bit	00c3f9a03daff818b2132ac31d57f054925c60e7
go1.4.2.darwin-amd64-osx10.8.tar.gz	Archive	OS X 10.8+	64-bit	58a04b3eb9853c75319d9076df6f3ac8b7430f7f
go1.4.2.darwin-amd64-osx10.6.pkg	Installer	OS X 10.6+	64-bit	3fa5455e211a70c0a920abd53cb3093269c5149c
go1.4.2.darwin-amd64-osx10.8.pkg	**Installer**	**OS X 10.8+**	**64-bit**	**8fde619d48864cb1c77ddc2a1aec0b7b20406b38**
go1.4.2.linux-386.tar.gz	Archive	Linux	32-bit	50557248e89b6e38d395fda93b2f96b2b860a26a
go1.4.2.linux-amd64.tar.gz	**Archive**	**Linux**	**64-bit**	**5020af94b52b65cc9b6f11d50a67e4bae07b0aff**
go1.4.2.windows-386.zip	Archive	Windows	32-bit	0e074e66a7816561d7947ff5c3514be96f347dc4
go1.4.2.windows-386.msi	Installer	Windows	32-bit	e8bd3d87cb52441b2c9aee7c2c5f5ce7ffccc832
go1.4.2.windows-amd64.zip	Archive	Windows	64-bit	91b229a3ff0a1ce6e791c832b0b4670bfc5457b5
go1.4.2.windows-amd64.msi	**Installer**	**Windows**	**64-bit**	**a914f3dad5521a8f658dce3e1575f3b6792975f0**

UNIX/Linux/Mac OS X, 和 FreeBSD 安装

以下介绍了在UNIX/Linux/Mac OS X, 和 FreeBSD系统下使用源码安装方法：

1、 下载源码包：go1.4.linux-amd64.tar.gz。

2、将下载的源码包解压至 /usr/local目录。

```
tar -C /usr/local -xzf go1.4.linux-amd64.tar.gz
```

3、将 /usr/local/go/bin 目录添加至PATH环境变量：

```
export PATH=$PATH:/usr/local/go/bin
```

注意：MAC 系统下你可以使用 .pkg 结尾的安装包直接双击来完成安装，安装目录在 /usr/local/go/ 下。

Windows 系统下安装

Windows 下可以使用 .msi 后缀(在下载列表中可以找到该文件，如 go1.4.2.windows-amd64.msi)的安装包来安装。

默认情况下.msi文件会安装在 c:\Go 目录下。你可以将 c:\Go\bin 目录添加到 PATH 环境变量中。添加后你需要重启命令窗口才能生效。

安装测试

创建工作目录 C:\>Go_WorkSpace。

文件名: test.go，代码如下：

```
package main

import "fmt"

func main() {
   fmt.Println("Hello, World!")
}
```

使用 go 命令执行以上代码输出结果如下：

```
C:\Go_WorkSpace>go run test.go

Hello, World!
```

Go 语言结构

在我们开始学习 GO 编程语言的基础构建模块前，让我们先来了解 Go 语言最简单程序的结构。

Go Hello World 实例

Go 语言的基础组成有以下几个部分：

- 包声明
- 引入包
- 函数
- 变量
- 语句 & 表达式
- 注释

接下来让我们来看下简单的代码，该代码输出了"Hello World!":

```
package main import  "fmt" func main()  {  /* 这是我的第一个简单的程序
```

让我们来看下以上程序的各个部分：

1. 第一行代码 *package main* 定义了包名。你必须在源文件中非注释的第一行指明这个文件属于哪个包，如：package main。package main表示一个可独立执行的程序，每个 Go 应用程序都包含一个名为 main 的包。
2. 下一行 *import "fmt"* 告诉 Go 编译器这个程序需要使用 fmt 包（的函数，或其他元素），fmt 包实现了格式化 IO（输入/输出）的函数。
3. 下一行 *func main()* 是程序开始执行的函数。main 函数是每一个可执行程序所必须包含的，一般来说都是在启动后第一个执行的函数（如果有 init() 函数则会先执行该函数）。
4. 下一行 /.../ 是注释，在程序执行时将被忽略。单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 / 开头，并以 / 结尾，且不可以嵌套使用，多行注释一般用于包的文档描述或注释成块的代码片段。
5. 下一行 *fmt.Println(...)* 可以将字符串输出到控制台，并在最后自动增加换行字符 \n。 使用 fmt.Print("hello, world\n") 可以得到相同的结果。 Print 和 Println 这两个函数也支持使用变量，如：fmt.Println(arr)。如果没有特别指定，它们会以默认的打印格式将变量 arr 输出到控制台。

6. 当标识符（包括常量、变量、类型、函数名、结构字段等等）以一个大写字母开头，如：Group1，那么使用这种形式的标识符的对象就可以被外部包的代码所使用（客户端程序需要先导入这个包），这被称为导出（像面向对象语言中的 public）；标识符如果以小写字母开头，则对包外是不可见的，但是他们在整个包的内部是可见并且可用的（像面向对象语言中的 private ）。

执行 Go 程序

让我们来看下如何编写 Go 代码并执行它。步骤如下：

1. 打开编辑器如Sublime2，将以上代码添加到编辑器中。
2. 将以上代码保存为 *hello.go*
3. 打开命令行，并进入程序文件保存的目录中。
4. 输入命令 *go run hello.go* 并按回车执行代码。
5. 如果操作正确你将在屏幕上看到 *"Hello World!"* 字样的输出。

```
$ go run hello.go Hello,  World!
```

Go 语言基础语法

上一章节我们已经了解了 Go 语言的基本组成结构，本章节我们将学习 Go 语言的基础语法。

Go 标记

Go 程序可以由多个标记组成，可以是关键字，标识符，常量，字符串，符号。如以下 GO 语句由 6 个标记组成：

```
fmt.Println("Hello, World!")
```

6 个标记是(每行一个)：

```
1\. fmt
2\. .
3\. Println
4\. (
5\. "Hello, World!"
6\. )
```

行分隔符

在 Go 程序中，一行代表一个语句结束。每个语句不需要像 C 家族中的其它语言一样以分号 ; 结尾，因为这些工作都将由 Go 编译器自动完成。

如果你打算将多个语句写在同一行，它们则必须使用 ; 人为区分，但在实际开发中我们并不鼓励这种做法。

以下为两个语句：

```
fmt.Println("Hello, World!")
fmt.Println("w3cschool菜鸟教程：w3cschool.cc")
```

注释

注释不会被编译，每一个包应该有相关注释。

单行注释是最常见的注释形式，你可以在任何地方使用以 // 开头的单行注释。多行注释也叫块注释，均已以 / 开头，并以 / 结尾。如：

```
// 单行注释
/*
 Author by w3cschool菜鸟教程
 我是多行注释
 */
```

标识符

标识符用来命名变量、类型等程序实体。一个标识符实际上就是一个或是多个字母(A~Z和a~z)数字(0~9)、下划线_组成的序列，但是第一个字符必须是字母或下划线而不能是数字。

以下是有效的标识符：

```
mahesh   kumar   abc   move_name   a_123
myname50   _temp   j   a23b9   retVal
```

以下是无效的标识符：

- 1ab（以数字开头）
- case（Go 语言的关键字）
- a+b（运算符是不允许的）

关键字

下面列举了 Go 代码中会使用到的 25 个关键字或保留字：

break	default	func	interface	select
case	defer	go	map	struct
chan	else	goto	package	switch
const	fallthrough	if	range	type
continue	for	import	return	var

除了以上介绍的这些关键字，Go 语言还有 36 个预定义标识符：

append	bool	byte	cap	close	complex	complex64	co
copy	false	float32	float64	imag	int	int8	in
int32	int64	iota	len	make	new	nil	pa
print	println	real	recover	string	true	uint	ui

程序一般由关键字、常量、变量、运算符、类型和函数组成。

程序中可能会使用到这些分隔符：括号 ()，中括号 [] 和大括号 {}。

程序中可能会使用到这些标点符号：.、,、;、: 和 …。

Go 语言的空格

Go 语言中变量的声明必须使用空格隔开，如：

```
var age int;
```

语句中适当使用空格能让程序看易阅读。

无空格：

```
fruit=apples+oranges;
```

在变量与运算符间加入空格，程序看起来更加美观，如：

```
fruit = apples + oranges;
```

Go 语言数据类型

在 Go 编程语言中，数据类型用于声明函数和变量。

数据类型的出现是为了把数据分成所需内存大小不同的数据，编程的时候需要用大数据的时候才需要申请大内存，就可以充分利用内存。

Go 语言按类别有以下几种数据类型：

序号	类型和描述
布尔型	布尔型的值只可以是常量 true 或者 false。一个简单的例子：var b bool = true。
数字类型	整型 int 和浮点型 float，Go 语言支持整型和浮点型数字，并且原生支持复数，其中位的运算采用补码。
字符串类型:	字符串就是一串固定长度的字符连接起来的字符序列。Go的字符串是由单个字节连接起来的。Go语言的字符串的字节使用UTF-8编码标识Unicode文本。
派生类型:	包括：(a) 指针类型（Pointer） (b) 数组类型 (c) 结构化类型(struct) (d) 联合体类型 (union) (e) 函数类型 (f) 切片类型 (g) 接口类型（interface） (h) Map 类型 (i) Channel 类型

数字类型

Go 也有基于架构的类型，例如：int、uint 和 uintptr。

序号	类型和描述
uint8	无符号 8 位整型 (0 到 255)
uint16	无符号 16 位整型 (0 到 65535)
uint32	无符号 32 位整型 (0 到 4294967295)
uint64	无符号 64 位整型 (0 到 18446744073709551615)
int8	有符号 8 位整型 (-128 到 127)
int16	有符号 16 位整型 (-32768 到 32767)
int32	有符号 32 位整型 (-2147483648 到 2147483647)
int64	有符号 64 位整型 (-9223372036854775808 到 9223372036854775807)

浮点型：

序号	类型和描述
float32	IEEE-754 32位浮点型数
float64	IEEE-754 64位浮点型数
complex64	32 位实数和虚数
complex128	64 位实数和虚数

其他数字类型

以下列出了其他更多的数字类型：

序号	类型和描述
byte	类似 uint8
rune	类似 int32
uint	32 或 64 位
int	与 uint 一样大小
uintptr	无符号整型，用于存放一个指针

Go 语言变量

变量来源于数学，是计算机语言中能储存计算结果或能表示值抽象概念。变量可以通过变量名访问。

Go 语言变量名由字母、数字、下划线组成，其中首个字母不能为数字。

声明变量的一般形式是使用 var 关键字：

```
var identifier type
```

变量声明

第一种，指定变量类型，声明后若不赋值，使用默认值。

```
var v_name v_type
v_name = value
```

第二种，根据值自行判定变量类型。

```
var v_name = value
```

第三种，省略var, 注意 :=左侧的变量不应该是已经声明过的，否则会导致编译错误。

```
v_name := value

// 例如
var a int = 10
var b = 10
c : = 10
```

实例如下：

```
package main
var a = "w3cschool菜鸟教程"
var b string = "w3cschool.cc"
var c bool

func main(){
    println(a, b, c)
}
```

以上实例执行结果为：

```
w3cschool菜鸟教程 w3cschool.cc false
```

多变量声明

```
//类型相同多个变量, 非全局变量
var vname1, vname2, vname3 type
vname1, vname2, vname3 = v1, v2, v3

var vname1, vname2, vname3 = v1, v2, v3 //和python很像,不需要显示声明类

vname1, vname2, vname3 := v1, v2, v3 //出现在:=左侧的变量不应该是已经被声

// 这种因式分解关键字的写法一般用于声明全局变量
var (
    vname1 v_type1
    vname2 v_type2
)
```

实例如下：

```
package main

var x, y int
var (  // 这种因式分解关键字的写法一般用于声明全局变量
    a int
    b bool
)

var c, d int = 1, 2
var e, f = 123, "hello"

//这种不带声明格式的只能在函数体中出现
//g, h := 123, "hello"

func main(){
    g, h := 123, "hello"
    println(x, y, a, b, c, d, e, f, g, h)
}
```

以上实例执行结果为：

```
0 0 0 false 1 2 123 hello 123 hello
```

值类型和引用类型

所有像 int、float、bool 和 string 这些基本类型都属于值类型，使用这些类型的变量直接指向存在内存中的值：

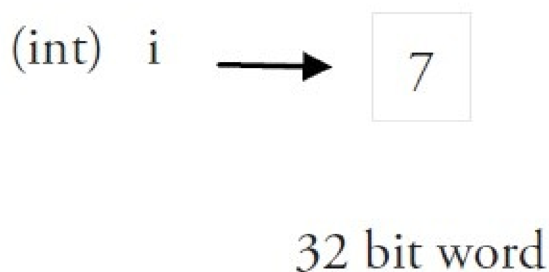


Fig 4.1: Value type

当使用等号 `=` 将一个变量的值赋值给另一个变量时，如：`j = i`，实际上是在内存中将 i 的值进行了拷贝：

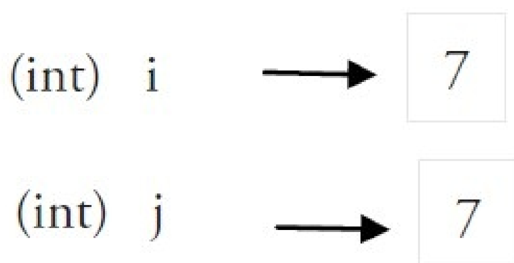


Fig 4.2: Assignment of value types

你可以通过 &i 来获取变量 i 的内存地址，例如：0xf840000040（每次的地址都可能不一样）。值类型的变量的值存储在栈中。

内存地址会根据机器的不同而有所不同，甚至相同的程序在不同的机器上执行后也会有不同的内存地址。因为每台机器可能有不同的存储器布局，并且位置分配也可能不同。

更复杂的数据通常会需要使用多个字，这些数据一般使用引用类型保存。

一个引用类型的变量 r1 存储的是 r1 的值所在的内存地址（数字），或内存地址中第一个字所在的位置。



Fig 4.3: Reference types and assignment

这个内存地址为称之为指针，这个指针实际上也被存在另外的某一个字中。

同一个引用类型的指针指向的多个字可以是在连续的内存地址中（内存布局是连续的），这也是计算效率最高的一种存储形式；也可以将这些字分散存放在内存中，每个字都指示了下一个字所在的内存地址。

当使用赋值语句 r2 = r1 时，只有引用（地址）被复制。

如果 r1 的值被改变了，那么这个值的所有引用都会指向被修改后的内容，在这个例子中，r2 也会受到影响。

简短形式，使用 := 赋值操作符

我们知道可以在变量的初始化时省略变量的类型而由系统自动推断，声明语句写上 var 关键字其实是显得有些多余了，因此我们可以将它们简写为 a := 50 或 b := false。

a 和 b 的类型（int 和 bool）将由编译器自动推断。

这是使用变量的首选形式，但是它只能被用在函数体内，而不可以用于全局变量的声明与赋值。使用操作符 := 可以高效地创建一个新的变量，称之为初始化声明。

注意事项

如果在相同的代码块中，我们不可以再次对于相同名称的变量使用初始化声明，例如：a := 20 就是不被允许的，编译器会提示错误 no new variables on left side of :=，但是 a = 20 是可以的，因为这是给相同的变量赋予一个新的值。

如果你在定义变量 a 之前使用它，则会得到编译错误 undefined: a。

如果你声明了一个局部变量却没有在相同的代码块中使用它，同样会得到编译错误，例如下面这个例子当中的变量 a：

```
func main() {
   var a string = "abc"
   fmt.Println("hello, world")
}
```

尝试编译这段代码将得到错误 **a declared and not used**。

此外，单纯地给 a 赋值也是不够的，这个值必须被使用，所以使用

```
fmt.Println("hello, world", a)
```

会移除错误。

但是全局变量是允许声明但不使用。

>

同一类型的多个变量可以声明在同一行，如：

```
var a, b, c int
```

多变量可以在同一行进行赋值，如：

```
a, b, c = 5, 7, "abc"
```

上面这行假设了变量 a，b 和 c 都已经被声明，否则的话应该这样使用：

```
a, b, c := 5, 7, "abc"
```

右边的这些值以相同的顺序赋值给左边的变量，所以 a 的值是 5， b 的值是 7，c 的值是 "abc"。

这被称为 并行 或 同时 赋值。

如果你想要交换两个变量的值，则可以简单地使用 a, b = b, a。

空白标识符 _ 也被用于抛弃值，如值 5 在：_, b = 5, 7 中被抛弃。

_ 实际上是一个只写变量，你不能得到它的值。这样做是因为 Go 语言中你必须使用所有被声明的变量，但有时你并不需要使用从一个函数得到的所有返回值。

并行赋值也被用于当一个函数返回多个返回值时，比如这里的 val 和错误 err 是通过调用 Func1 函数同时得到：val, err = Func1(var1)。

Go 语言常量

常量是一个简单值的标识符，在程序运行时，不会被修改的量。

常量中的数据类型只可以是布尔型、数字型（整数型、浮点型和复数）和字符串型。

常量的定义格式：

```
const identifier [type] = value
```

你可以省略类型说明符 [type]，因为编译器可以根据变量的值来推断其类型。

- 显式类型定义： `const b string = "abc"`
- 隐式类型定义： `const b = "abc"`

多个相同类型的声明可以简写为：

```
const c_name1, c_name2 = value1, value2
```

以下实例演示了常量的应用：

```
package main

import "fmt"

func main() {
   const LENGTH int = 10
   const WIDTH int = 5
   var area int
   const a, b, c = 1, false, "str" //多重赋值

   area = LENGTH * WIDTH
   fmt.Printf("面积为 : %d", area)
   println()
   println(a, b, c)
}
```

以上实例运行结果为：

```
面积为 : 50
1 false str
```

常量还可以用作枚举：

```
const (
    Unknown = 0
    Female = 1
    Male = 2
)
```

数字 0、1 和 2 分别代表未知性别、女性和男性。

常量可以用len(), cap(), unsafe.Sizeof()常量计算表达式的值。常量表达式中，函数必须是内置函数，否则编译不过：

```
package main

import "unsafe"
const (
    a = "abc"
    b = len(a)
    c = unsafe.Sizeof(a)
)

func main(){
    println(a, b, c)
}
```

以上实例运行结果为：

```
abc 3 16
```

iota

iota，特殊常量，可以认为是一个可以被编译器修改的常量。

在每一个const关键字出现时，被重置为0，然后再下一个const出现之前，每出现一次iota，其所代表的数字会自动增加1。

iota 可以被用作枚举值：

```
const (
    a = iota
    b = iota
    c = iota
)
```

第一个 iota 等于 0，每当 iota 在新的一行被使用时，它的值都会自动加 1；所以 a=0, b=1, c=2 可以简写为如下形式：

```
const (
    a = iota
    b
    c
)
```

iota 用法

```
package main

import "fmt"

func main() {
    const (
            a = iota   //0
            b          //1
            c          //2
            d = "ha"   //独立值, iota += 1
            e          //"ha"   iota += 1
            f = 100    //iota +=1
            g          //100  iota +=1
            h = iota   //7,恢复计数
            i          //8
    )
    fmt.Println(a,b,c,d,e,f,g,h,i)
}
```

以上实例运行结果为：

```
0 1 2 ha ha 100 100 7 8
```

再看个有趣的的 iota 实例：

```
package main

import "fmt"
const (
    i=1<<iota
    j=3<<iota
    k
    l
)

func main() {
    fmt.Println("i=",i)
    fmt.Println("j=",j)
    fmt.Println("k=",k)
    fmt.Println("l=",l)
}
```

以上实例运行结果为：

```
i= 1
j= 6
k= 12
l= 24
```

iota表示从0开始自动加1，所以i=1<<0,j=3<<1（<<表示左移的意思），即：i=1,j=6，这没问题，关键在k和l，从输出结果看，k=3<<2，l=3<<3。

Go 语言运算符

运算符用于在程序运行时执行数学或逻辑运算。

Go 语言内置的运算符有：

- 算术运算符
- 关系运算符
- 逻辑运算符
- 位运算符
- 赋值运算符
- 其他运算符

接下来让我们来详细看看各个运算符的介绍。

算术运算符

下表列出了所有Go语言的算术运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
+	相加	A + B 输出结果 30
-	相减	A - B 输出结果 -10
*	相乘	A * B 输出结果 200
/	相除	B / A 输出结果 2
%	求余	B % A 输出结果 0
++	自增	A++ 输出结果 11
--	自减	A-- 输出结果 9

以下实例演示了各个算术运算符的用法：

```
package main

import "fmt"

func main() {

   var a int = 21
   var b int = 10
   var c int

   c = a + b
   fmt.Printf("第一行 - c 的值为 %d\n", c )
   c = a - b
   fmt.Printf("第二行 - c 的值为 %d\n", c )
   c = a * b
   fmt.Printf("第三行 - c 的值为 %d\n", c )
   c = a / b
   fmt.Printf("第四行 - c 的值为 %d\n", c )
   c = a % b
   fmt.Printf("第五行 - c 的值为 %d\n", c )
   a++
   fmt.Printf("第六行 - c 的值为 %d\n", a )
   a--
   fmt.Printf("第七行 - c 的值为 %d\n", a )
}
```

以上实例运行结果：

```
第一行 - c 的值为 31
第二行 - c 的值为 11
第三行 - c 的值为 210
第四行 - c 的值为 2
第五行 - c 的值为 1
第六行 - c 的值为 22
第七行 - c 的值为 21
```

关系运算符

下表列出了所有Go语言的关系运算符。假定 A 值为 10，B 值为 20。

运算符	描述	实例
==	检查两个值是否相等，如果相等返回 True 否则返回 False。	(A == B) 为 False
!=	检查两个值是否不相等，如果不相等返回 True 否则返回 False。	(A != B) 为 True
>	检查左边值是否大于右边值，如果是返回 True 否则返回 False。	(A > B) 为 False
<	检查左边值是否小于右边值，如果是返回 True 否则返回 False。	(A < B) 为 True
>=	检查左边值是否大于等于右边值，如果是返回 True 否则返回 False。	(A >= B) 为 False
<=	检查左边值是否小于等于右边值，如果是返回 True 否则返回 False。	(A <= B) 为 True

以下实例演示了关系运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 21
   var b int = 10

   if( a == b ) {
      fmt.Printf("第一行 - a 等于 b\n" )
   } else {
      fmt.Printf("第一行 - a 不等于 b\n" )
   }
   if ( a < b ) {
      fmt.Printf("第二行 - a 小于 b\n" )
   } else {
      fmt.Printf("第二行 - a 不小于 b\n" )
   }

   if ( a > b ) {
      fmt.Printf("第三行 - a 大于 b\n" )
   } else {
      fmt.Printf("第三行 - a 不大于 b\n" )
   }
   /* Lets change value of a and b */
   a = 5
   b = 20
   if ( a <= b ) {
      fmt.Printf("第四行 - a 小于等于  b\n" )
   }
   if ( b >= a ) {
      fmt.Printf("第五行 - b 大于等于 b\n" )
   }
}
```

以上实例运行结果：

```
第一行 - a 不等于 b
第二行 - a 不小于 b
第三行 - a 大于 b
第四行 - a 小于等于  b
第五行 - b 大于等于 b
```

逻辑运算符

下表列出了所有Go语言的逻辑运算符。假定 A 值为 True，B 值为 False。

运算符	描述	实例
&&	逻辑 AND 运算符。 如果两边的操作数都是 True，则条件 True，否则为 False。	(A && B) 为 False
||	逻辑 OR 运算符。 如果两边的操作数有一个 True，则条件 True，否则为 False。	(A || B) 为 True
!	逻辑 NOT 运算符。 如果条件为 True，则逻辑 NOT 条件 False，否则为 True。	!(A && B) 为 True

以下实例演示了逻辑运算符的用法：

```
package main

import "fmt"

func main() {
   var a bool = true
   var b bool = false
   if ( a && b ) {
      fmt.Printf("第一行 - 条件为 true\n" )
   }
   if ( a || b ) {
      fmt.Printf("第二行 - 条件为 true\n" )
   }
   /* 修改 a 和 b 的值 */
   a = false
   b = true
   if ( a && b ) {
      fmt.Printf("第三行 - 条件为 true\n" )
   } else {
      fmt.Printf("第三行 - 条件为 false\n" )
   }
   if ( !(a && b) ) {
      fmt.Printf("第四行 - 条件为 true\n" )
   }
}
```

以上实例运行结果：

```
第二行 - 条件为 true
第三行 - 条件为 false
第四行 - 条件为 true
```

位运算符

位运算符对整数在内存中的二进制位进行操作。

下表列出了位运算符 &, |, 和 ^ 的计算：

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

假定 A = 60; B = 13; 其二进制数转换为：

```
A = 0011 1100

B = 0000 1101

-----------------

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

~A  = 1100 0011
```

C 语言支持的位运算符如下表所示。假定 A 为60，B 为13：

<table>
<tr><th>运算符</th><th>描述</th><th>实例</th></tr>
<tr><td>&</td><td>按位与运算符"&"是双目运算符。 其功能是参与运算的两数各对应的二进位相与。</td><td>(A & B) 结果为 12, 二进制为 0000 1100</td></tr>
<tr><td>|</td><td>按位或运算符"|"是双目运算符。 其功能是参与运算的两数各对应的二进位相或</td><td>(A | B) 结果为 61, 二进制为 0011 1101</td></tr>
<tr><td>^</td><td>按位异或运算符"^"是双目运算符。 其功能是参与运算的两数各对应的二进位相异或，当两对应的二进位相异时，结果为1。</td><td>(A ^ B) 结果为 49, 二进制为 0011 0001</td></tr>
<tr><td><<</td><td>左移运算符"<<"是双目运算符。左移n位就是乘以2的n次方。 其功能把"<<"左边的运算数的各二进位全部左移若干位，由"<<"右边的数指定移动的位数，高位丢弃，低位补0。</td><td>A << 2 结果为 240 ，二进制为 1111 0000</td></tr>
<tr><td>>></td><td>右移运算符">>"是双目运算符。右移n位就是除以2的n次方。 其功能是把">>"左边的运算数的各二进位全部右移若干位，">>"右边的数指定移动的位数。</td><td>A >> 2 结果为 15 ，二进制为 0000 1111</td></tr>
</table>

以下实例演示了逻辑运算符的用法：

```
package main

import "fmt"

func main() {

   var a uint = 60    /* 60 = 0011 1100 */
   var b uint = 13    /* 13 = 0000 1101 */
   var c uint = 0

   c = a & b       /* 12 = 0000 1100 */
   fmt.Printf("第一行 - c 的值为 %d\n", c )

   c = a | b       /* 61 = 0011 1101 */
   fmt.Printf("第二行 - c 的值为 %d\n", c )

   c = a ^ b       /* 49 = 0011 0001 */
   fmt.Printf("第三行 - c 的值为 %d\n", c )

   c = a << 2     /* 240 = 1111 0000 */
   fmt.Printf("第四行 - c 的值为 %d\n", c )

   c = a >> 2     /* 15 = 0000 1111 */
   fmt.Printf("第五行 - c 的值为 %d\n", c )
}
```

以上实例运行结果：

```
第一行 - c 的值为 12
第二行 - c 的值为 61
第三行 - c 的值为 49
第四行 - c 的值为 240
第五行 - c 的值为 15
```

赋值运算符

下表列出了所有Go语言的赋值运算符。

运算符	描述	实例
=	简单的赋值运算符，将一个表达式的值赋给一个左值	C = A + B 将 A + B 表达式结果赋值给 C
+=	相加后再赋值	C += A 等于 C = C + A
-=	相减后再赋值	C -= A 等于 C = C - A
*=	相乘后再赋值	C = A 等于 C = C A
/=	相除后再赋值	C /= A 等于 C = C / A
%=	求余后再赋值	C %= A 等于 C = C % A
<<=	左移赋值	C <<= 2 等于 C = C << 2
>>=	右移赋值	C >>= 2 等于 C = C >> 2
&=	位逻辑与赋值	C &= 2 等于 C = C & 2
^=	位逻辑或赋值	C ^= 2 等于 C = C ^ 2
|=	位逻辑异或赋值	C |= 2 等于 C = C | 2

以下实例演示了赋值运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 21
   var c int

   c =  a
   fmt.Printf("第 1 行 - =  运算符实例，c 值为 = %d\n", c )

   c +=  a
   fmt.Printf("第 2 行 - += 运算符实例，c 值为 = %d\n", c )

   c -=  a
   fmt.Printf("第 3 行 - -= 运算符实例，c 值为 = %d\n", c )

   c *=  a
   fmt.Printf("第 4 行 - *= 运算符实例，c 值为 = %d\n", c )

   c /=  a
   fmt.Printf("第 5 行 - /= 运算符实例，c 值为 = %d\n", c )

   c  = 200;

   c <<=  2
   fmt.Printf("第 6行  - <<= 运算符实例，c 值为 = %d\n", c )

   c >>=  2
   fmt.Printf("第 7 行 - >>= 运算符实例，c 值为 = %d\n", c )

   c &=  2
   fmt.Printf("第 8 行 - &= 运算符实例，c 值为 = %d\n", c )

   c ^=  2
   fmt.Printf("第 9 行 - ^= 运算符实例，c 值为 = %d\n", c )

   c |=  2
   fmt.Printf("第 10 行 - |= 运算符实例，c 值为 = %d\n", c )

}
```

以上实例运行结果：

```
第 1 行 - =  运算符实例，c 值为 = 21
第 2 行 - += 运算符实例，c 值为 = 42
第 3 行 - -= 运算符实例，c 值为 = 21
第 4 行 - *= 运算符实例，c 值为 = 441
第 5 行 - /= 运算符实例，c 值为 = 21
第 6行  - <<= 运算符实例，c 值为 = 800
第 7 行 - >>= 运算符实例，c 值为 = 200
第 8 行 - &= 运算符实例，c 值为 = 0
第 9 行 - ^= 运算符实例，c 值为 = 2
第 10 行 - |= 运算符实例，c 值为 = 2
```

其他运算符

下表列出了Go语言的其他运算符。

运算符	描述	实例
&	返回变量存储地址	&a; 将给出变量的实际地址。
*	指针变量。	*a; 是一个指针变量

以下实例演示了其他运算符的用法：

```
package main

import "fmt"

func main() {
   var a int = 4
   var b int32
   var c float32
   var ptr *int

   /* 运算符实例 */
   fmt.Printf("第 1 行 - a 变量类型为 = %T\n", a );
   fmt.Printf("第 2 行 - b 变量类型为 = %T\n", b );
   fmt.Printf("第 3 行 - c 变量类型为 = %T\n", c );

   /*  & 和 * 运算符实例 */
   ptr = &a    /* 'ptr' 包含了 'a' 变量的地址 */
   fmt.Printf("a 的值为  %d\n", a);
   fmt.Printf("*ptr 为 %d\n", *ptr);
}
```

以上实例运行结果：

```
第 1 行 - a 变量类型为 = int
第 2 行 - b 变量类型为 = int32
第 3 行 - c 变量类型为 = float32
a 的值为  4
*ptr 为 4
```

运算符优先级

有些运算符拥有较高的优先级，二元运算符的运算方向均是从左至右。下表列出了所有运算符以及它们的优先级，由上至下代表优先级由高到低：

优先级	运算符
7	^ !
6	* / % << >> & &^
5	+ - | ^
4	== != < <= >= >
3	<-
2	&&
1	||

当然，你可以通过使用括号来临时提升某个表达式的整体运算优先级。

以上实例运行结果：

```
package main

import "fmt"

func main() {
   var a int = 20
   var b int = 10
   var c int = 15
   var d int = 5
   var e int;

   e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   fmt.Printf("(a + b) * c / d 的值为 : %d\n",  e );

   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   fmt.Printf("((a + b) * c) / d 的值为  : %d\n" ,  e );

   e = (a + b) * (c / d);   // (30) * (15/5)
   fmt.Printf("(a + b) * (c / d) 的值为  : %d\n",  e );

   e = a + (b * c) / d;     //  20 + (150/5)
   fmt.Printf("a + (b * c) / d 的值为  : %d\n" ,  e );
}
```

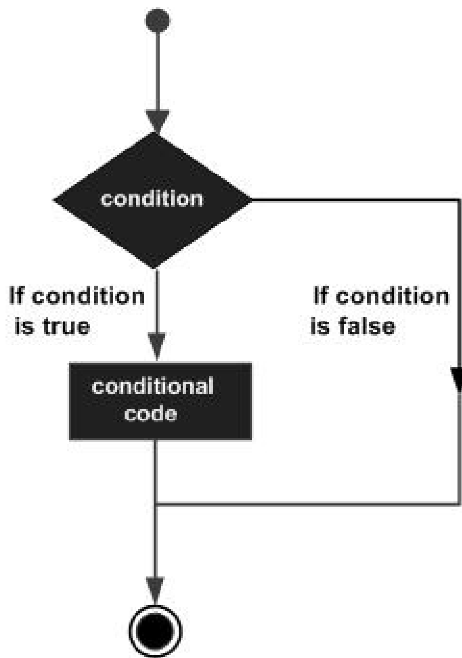
以上实例运行结果：

```
(a + b) * c / d 的值为 : 90
((a + b) * c) / d 的值为  : 90
(a + b) * (c / d) 的值为  : 90
a + (b * c) / d 的值为  : 50
```

Go 语言条件语句

条件语句需要开发者通过指定一个或多个条件，并通过测试条件是否为 true 来决定是否执行指定语句，并在条件为 false 的情况在执行另外的语句。

下图展示了程序语言中条件语句的结构：



Go 语言提供了以下几种条件判断语句：

语句	描述
if 语句	**if 语句** 由一个布尔表达式后紧跟一个或多个语句组成。
if...else 语句	**if 语句** 后可以使用可选的 **else 语句**, else 语句中的表达式在布尔表达式为 false 时执行。
if 嵌套语句	你可以在 **if** 或 **else if** 语句中嵌入一个或多个 **if** 或 **else if** 语句。
switch 语句	**switch** 语句用于基于不同条件执行不同动作。
select 语句	**select** 语句类似于 **switch** 语句，但是select会随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。

Go 语言 if 语句

if 语句由布尔表达式后紧跟一个或多个语句组成。

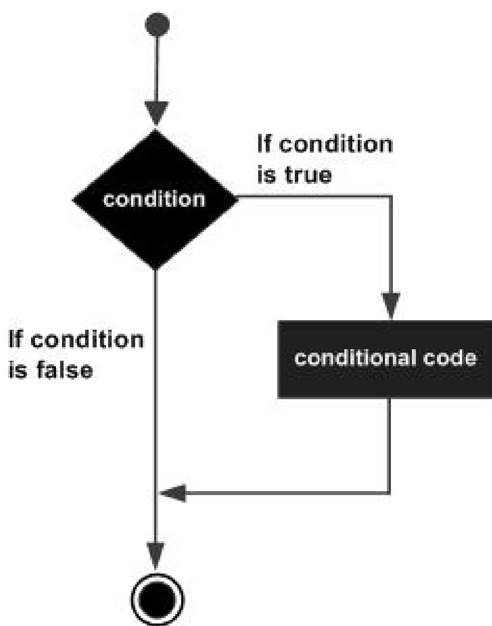
语法

Go 编程语言中 if 语句的语法如下：

```
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
}
```

If 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则不执行。

流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* 使用 if 语句判断布尔表达式 */
   if a < 20 {
       /* 如果条件为 true 则执行以下语句 */
       fmt.Printf("a 小于 20\n" )
   }
   fmt.Printf("a 的值为 : %d\n", a)
}
```

以上代码执行结果为：

```
a 小于  20
a 的值为  :  10
```

Go 语言 if...else 语句

if 语句 后可以使用可选的 else 语句, else 语句中的表达式在布尔表达式为 false 时执行。

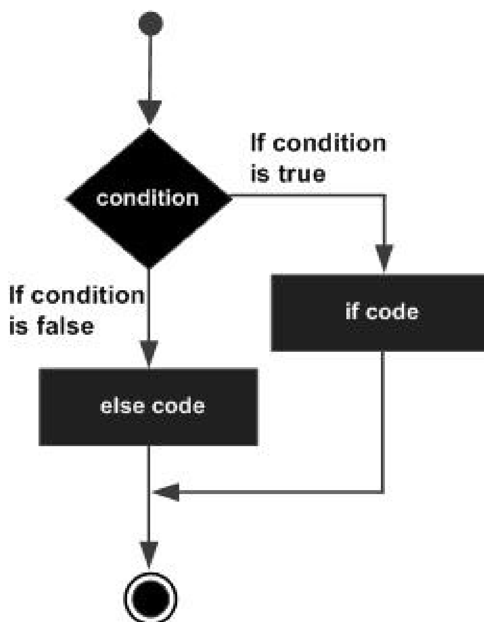
语法

Go 编程语言中 if...else 语句的语法如下：

```
if 布尔表达式 {
   /* 在布尔表达式为 true 时执行 */
} else {
  /* 在布尔表达式为 false 时执行 */
}
```

If 在布尔表达式为 true 时，其后紧跟的语句块执行，如果为 false 则执行 else 语句块。

流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 局部变量定义 */
   var a int = 100;

   /* 判断布尔表达式 */
   if a < 20 {
       /* 如果条件为 true 则执行以下语句 */
       fmt.Printf("a 小于 20\n" );
   } else {
       /* 如果条件为 false 则执行以下语句 */
       fmt.Printf("a 不小于 20\n" );
   }
   fmt.Printf("a 的值为 : %d\n", a);

}
```

以上代码执行结果为：

```
a 不小于  20
a 的值为  :  100
```

Go 语言 if 语句嵌套

你可以在 if 或 else if 语句中嵌入一个或多个 if 或 else if 语句。

语法

Go 编程语言中 if...else 语句的语法如下：

```
if 布尔表达式 1 {
   /* 在布尔表达式 1 为 true 时执行 */
   if 布尔表达式 2 {
      /* 在布尔表达式 2 为 true 时执行 */
   }
}
```

你可以以同样的方式在 if 语句中嵌套 **else if...else** 语句

实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200

   /* 判断条件 */
   if a == 100 {
       /* if 条件语句为 true 执行 */
       if b == 200 {
          /* if 条件语句为 true 执行 */
          fmt.Printf("a 的值为 100 ， b 的值为 200\n" );
       }
   }
   fmt.Printf("a 值为 : %d\n", a );
   fmt.Printf("b 值为 : %d\n", b );
}
```

以上代码执行结果为：

```
a 的值为  100 ， b 的值为  200
a 值为 : 100
b 值为 : 200
```

Go 语言 switch 语句

switch 语句用于基于不同条件执行不同动作，每一个 case 分支都是唯一的，从上直下逐一测试，直到匹配为止。。

switch 语句执行的过程从上至下，直到找到匹配项，匹配项后面也不需要再加 break

语法

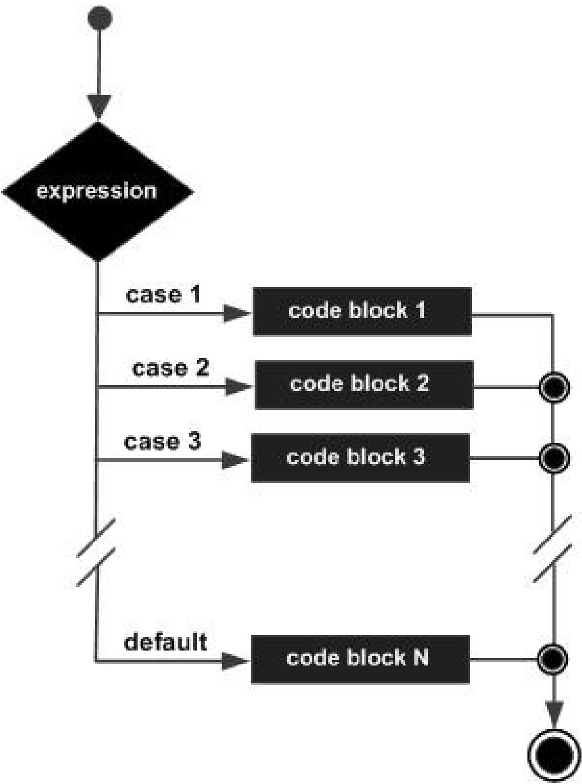
Go 编程语言中 switch 语句的语法如下：

```
switch var1 {
    case val1:
        ...
    case val2:
        ...
    default:
        ...
}
```

变量 var1 可以是任何类型，而 val1 和 val2 则可以是同类型的任意值。类型不被局限于常量或整数，但必须是相同的类型；或者最终结果为相同类型的表达式。

您可以同时测试多个可能符合条件的值，使用逗号分割它们，例如：case val1, val2, val3。

流程图：


expression
case 1
code block 1
case 2
code block 2
case 3
code block 3
default
code block N

实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var grade string = "B"
   var marks int = 90

   switch marks {
      case 90: grade = "A"
      case 80: grade = "B"
      case 50,60,70 : grade = "C"
      default: grade = "D"
   }

   switch {
      case grade == "A" :
         fmt.Printf("优秀!\n" )
      case grade == "B", grade == "C" :
         fmt.Printf("良好\n" )
      case grade == "D" :
         fmt.Printf("及格\n" )
      case grade == "F":
         fmt.Printf("不及格\n" )
      default:
         fmt.Printf("差\n" );
   }
   fmt.Printf("你的等级是 %s\n", grade );
}
```

以上代码执行结果为：

```
优秀!
你的等级是 A
```

Type Switch

switch 语句还可以被用于 type-switch 来判断某个 interface 变量中实际存储的变量类型。

Type Switch 语法格式如下：

```
switch x.(type){
    case type:
       statement(s);
    case type:
       statement(s);
    /* 你可以定义任意个数的case */
    default: /* 可选 */
       statement(s);
}
```

实例

```
package main

import "fmt"

func main() {
   var x interface{}

   switch i := x.(type) {
      case nil:
         fmt.Printf(" x 的类型 :%T",i)
      case int:
         fmt.Printf("x 是 int 型")
      case float64:
         fmt.Printf("x 是 float64 型")
      case func(int) float64:
         fmt.Printf("x 是 func(int) 型")
      case bool, string:
         fmt.Printf("x 是 bool 或 string 型" )
      default:
         fmt.Printf("未知型")
   }
}
```

以上代码执行结果为：

```
x 的类型  :<nil>
```

Go 语言 select 语句

select是Go中的一个控制结构，类似于用于通信的switch语句。每个case必须是一个通信操作，要么是发送要么是接收。

select随机执行一个可运行的case。如果没有case可运行，它将阻塞，直到有case可运行。一个默认的子句应该总是可运行的。

语法

Go 编程语言中 select 语句的语法如下：

```
select {
    case communication clause  :
       statement(s);
    case communication clause  :
       statement(s);
    /* 你可以定义任意数量的 case */
    default : /* 可选 */
       statement(s);
}
```

以下描述了 select 语句的语法：

- 每个case都必须是一个通信
- 所有channel表达式都会被求值
- 所有被发送的表达式都会被求值
- 如果任意某个通信可以进行，它就执行；其他被忽略。
- 如果有多个case都可以运行，Select会随机公平地选出一个执行。其他不会执行。 否则：
  1. 如果有default子句，则执行该语句。
  2. 如果没有default字句，select将阻塞，直到某个通信可以运行；Go不会重新对channel或值进行求值。

实例

```
package main

import "fmt"

func main() {
   var c1, c2, c3 chan int
   var i1, i2 int
   select {
      case i1 = <-c1:
         fmt.Printf("received ", i1, " from c1\n")
      case c2 <- i2:
         fmt.Printf("sent ", i2, " to c2\n")
      case i3, ok := (<-c3):  // same as: i3, ok := <-c3
         if ok {
            fmt.Printf("received ", i3, " from c3\n")
         } else {
            fmt.Printf("c3 is closed\n")
         }
      default:
         fmt.Printf("no communication\n")
   }
}
```

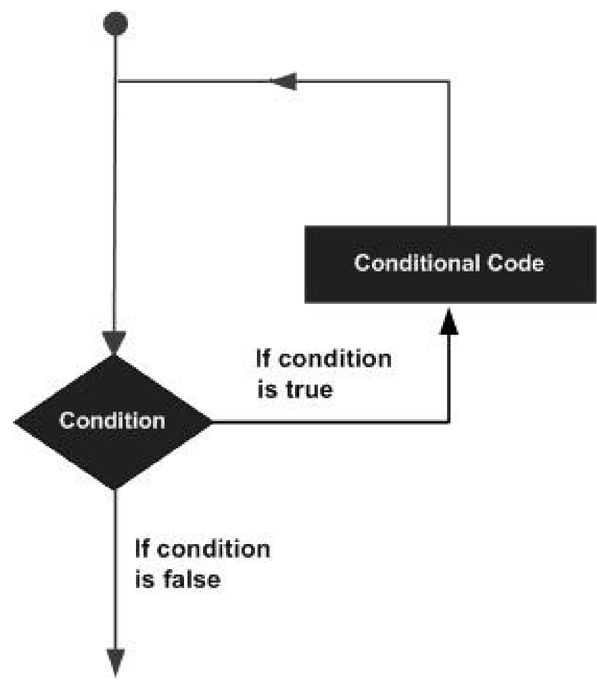
以上代码执行结果为：

```
no communication
```

Go 语言循环语句

在不少实际问题中有许多具有规律性的重复操作，因此在程序中就需要重复执行某些语句。

以下为大多编程语言循环程序的流程图：



Go 语言提供了以下几种类型循环处理语句：

循环类型	描述
for 循环	重复执行语句块
循环嵌套	在 for 循环中嵌套一个或多个 for 循环

循环控制语句

循环控制语句可以控制循环体内语句的执行过程。

GO 语言支持以下几种循环控制语句：

控制语句	描述
break 语句	经常用于中断当前 for 循环或跳出 switch 语句
continue 语句	跳过当前循环的剩余语句，然后继续进行下一轮循环。
goto 语句	将控制转移到被标记的语句。

无限循环

如过循环中条件语句永远不为 false 则会进行无限循环，我们可以通过 for 循环语句中只设置一个条件表达式来执行无限循环：

```
package main

import "fmt"

func main() {
    for true  {
        fmt.Printf("这是无限循环。\n");
    }
}
```

Go 语言 for 循环

for循环是一个循环控制结构，可以执行指定次数的循环。

语法

Go语言的For循环油3中形式，只有其中的一种使用分号。

和 C 语言的 for 一样：

```
for init; condition; post { }
```

和 C 的 while 一样：

```
for condition { }
```

和 C 的 for(;;) 一样：

```
for { }
```

- init： 一般为赋值表达式，给控制变量赋初值；
- condition： 关系表达式或逻辑表达式，循环控制条件；
- post： 一般为赋值表达式，给控制变量增量或减量。

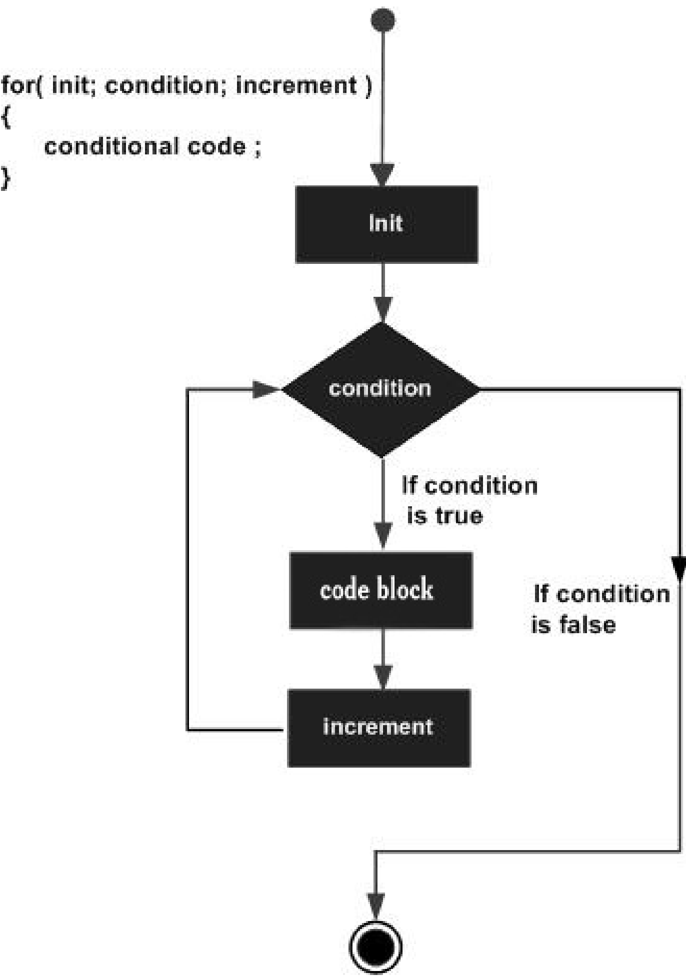
for语句执行过程如下：

- ①先对表达式1赋初值；
- ②判别赋值表达式 init 是否满足给定条件，若其值为真，满足循环条件，则执行循环体内语句，然后执行 post，进入第二次循环，再判别 condition；否则判断 condition 的值为假，不满足条件，就终止for循环，执行循环体外语句。

for 循环的 range 格式可以对 slice、map、数组、字符串等进行迭代循环。格式如下：

```
for key, value := range oldMap {
    newMap[key] = value
}
```

for语句语法流程如下图所示：


for(init; condition; increment)
{
conditional code ;
}
Init
condition
If condition
is true
code block
If condition
is false
increment

实例

```
package main

import "fmt"

func main() {

   var b int = 15
   var a int

   numbers := [6]int{1, 2, 3, 5}

   /* for 循环 */
   for a := 0; a < 10; a++ {
      fmt.Printf("a 的值为: %d\n", a)
   }

   for a < b {
      a++
      fmt.Printf("a 的值为: %d\n", a)
      }

   for i,x:= range numbers {
      fmt.Printf("第 %d 位 x 的值 = %d\n", i,x)
   }
}
```

以上实例运行输出结果为:

```
a 的值为: 0
a 的值为: 1
a 的值为: 2
a 的值为: 3
a 的值为: 4
a 的值为: 5
a 的值为: 6
a 的值为: 7
a 的值为: 8
a 的值为: 9
a 的值为: 1
a 的值为: 2
a 的值为: 3
a 的值为: 4
a 的值为: 5
a 的值为: 6
a 的值为: 7
a 的值为: 8
a 的值为: 9
a 的值为: 10
a 的值为: 11
a 的值为: 12
a 的值为: 13
a 的值为: 14
a 的值为: 15
第 0 位 x 的值 = 1
第 1 位 x 的值 = 2
第 2 位 x 的值 = 3
第 3 位 x 的值 = 5
第 4 位 x 的值 = 0
第 5 位 x 的值 = 0
```

Go 语言循环嵌套

Go 语言允许用户在循环内使用循环。接下来我们将为大家介绍嵌套循环的使用。

语法

以下为 Go 语言嵌套循环的格式：

```
for [condition |  ( init; condition; increment ) | Range]
{
   for [condition |  ( init; condition; increment ) | Range]
   {
      statement(s);
   }
   statement(s);
}
```

实例

以下实例使用循环嵌套来输出 2 到 100 间的素数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var i, j int

   for i=2; i < 100; i++ {
      for j=2; j <= (i/j); j++ {
         if(i%j==0) {
            break; // 如果发现因子，则不是素数
         }
      }
      if(j > (i/j)) {
         fmt.Printf("%d  是素数\n", i);
      }
   }
}
```

以上实例运行输出结果为:

```
2  是素数
3  是素数
5  是素数
7  是素数
11  是素数
13  是素数
17  是素数
19  是素数
23  是素数
29  是素数
31  是素数
37  是素数
41  是素数
43  是素数
47  是素数
53  是素数
59  是素数
61  是素数
67  是素数
71  是素数
73  是素数
79  是素数
83  是素数
89  是素数
97  是素数
```

Go 语言 break 语句

Go语言循环语句

Go 语言中 break 语句用于以下两方面：

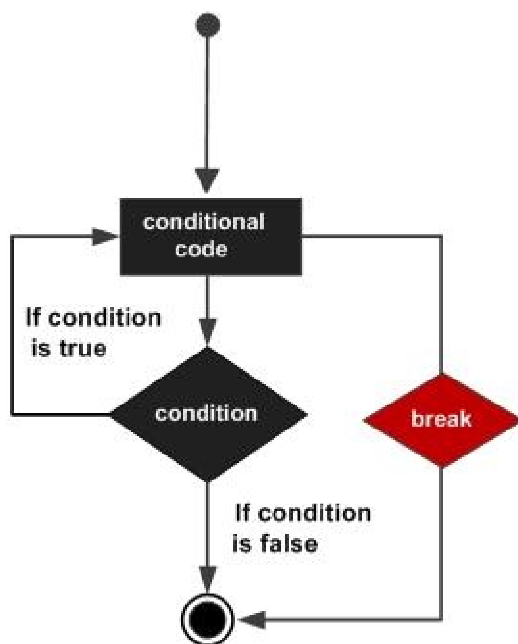
1. 用于循环语句中跳出循环，并开始执行循环之后的语句。
2. break在switch（开关语句）中在执行一条case后跳出语句的作用。

语法

break 语法格式如下：

```
break;
```

break 语句流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
      if a > 15 {
         /* 使用 break 语句跳出循环 */
         break;
      }
   }
}
```

以上实例执行结果为：

```
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 15
```

Go语言循环语句

Go 语言 continue 语句

Go 语言的 continue 语句 有点像 break 语句。但是 continue 不是跳出循环，而是跳过当前循环执行下一次循环语句。

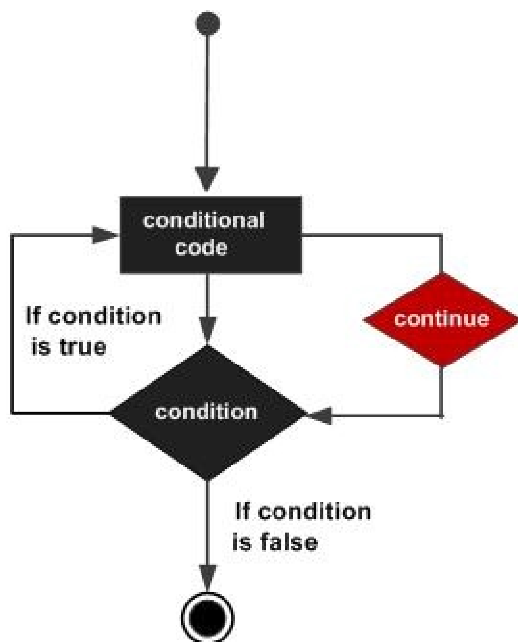
for 循环中，执行 continue 语句会触发for增量语句的执行。

语法

continue 语法格式如下：

```
continue;
```

break 语句流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* for 循环 */
   for a < 20 {
      if a == 15 {
         /* 跳过此次循环 */
         a = a + 1;
         continue;
      }
      fmt.Printf("a 的值为 : %d\n", a);
      a++;
   }
}
```

以上实例执行结果为：

```
a 的值为 : 10
a 的值为 : 10
a 的值为 : 11
a 的值为 : 12
a 的值为 : 13
a 的值为 : 14
a 的值为 : 16
a 的值为 : 17
a 的值为 : 18
a 的值为 : 19
```

Go 语言 goto 语句

Go 语言的 goto 语句可以无条件地转移到过程中指定的行。

goto语句通常与条件语句配合使用。可用来实现条件转移， 构成循环，跳出循环体等功能。

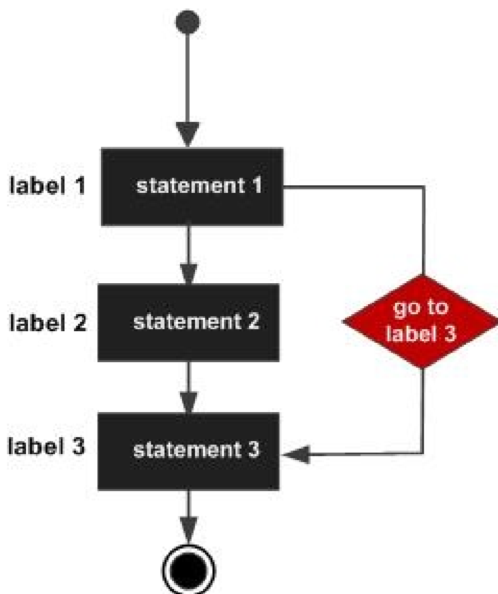
但是，在结构化程序设计中一般不主张使用goto语句， 以免造成程序流程的混乱，使理解和调试程序都产生困难。

语法

goto 语法格式如下：

```
goto label;
..
.
label: statement;
```

break 语句流程图如下：



实例

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 10

   /* 循环 */
   LOOP: for a < 20 {
      if a == 15 {
         /* 跳过迭代 */
         a = a + 1
         goto LOOP
      }
      fmt.Printf("a的值为 : %d\n", a)
      a++
   }
}
```

以上实例执行结果为：

```
a的值为 : 10
a的值为 : 11
a的值为 : 12
a的值为 : 13
a的值为 : 14
a的值为 : 16
a的值为 : 17
a的值为 : 18
a的值为 : 19
```

Go 语言函数

函数是基本的代码块，用于执行一个任务。

Go 语言最少有个 main() 函数。

你可以通过函数来划分不同功能，逻辑上每个函数执行的是指定的任务。

函数声明告诉了编译器函数的名称，返回类型，和参数。

Go 语言标准库提供了多种可动用的内置的函数。例如，len() 函数可以接受不同类型参数并返回该类型的长度。如果我们传入的是字符串则返回字符串的长度，如果传入的是数字，则返回数组中包含的函数个数。

函数定义

Go 语言函数定义格式如下：

```
func function_name( [parameter list] ) [return_types] {
   函数体
}
```

函数定义解析：

- func：函数由 func 开始声明
- function_name：函数名称，函数名和参数列表一起构成了函数签名。
- parameter list]：参数列表，参数就像一个占位符，当函数被调用时，你可以将值传递给参数，这个值被称为实际参数。参数列表指定的是参数类型、顺序、及参数个数。参数是可选的，也就是说函数也可以不包含参数。
- return_types：返回类型，函数返回一列值。return_types 是该列值的数据类型。有些功能不需要返回值，这种情况下 return_types 不是必须的。
- 函数体：函数定义的代码集合。

实例

以下实例为 max() 函数的代码，该函数传入两个整型参数 num1 和 num2，并返回这两个参数的最大值：

```
/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 声明局部变量 */
   result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

函数调用

当创建函数时，你定义了函数需要做什么，通过调用改函数来执行指定任务。

调用函数，向函数传递参数，并返回值，例如：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200
   var ret int

   /* 调用函数并返回最大值 */
   ret = max(a, b)

   fmt.Printf( "最大值是 : %d\n", ret )
}

/* 函数返回两个数的最大值 */
func max(num1, num2 int) int {
   /* 定义局部变量 */
   var result int

   if (num1 > num2) {
      result = num1
   } else {
      result = num2
   }
   return result
}
```

以上实例在 main() 函数中调用 max（）函数，执行结果为：

```
最大值是 : 200
```

函数返回多个值

Go 函数可以返回多个值，例如：

```
package main

import "fmt"

func swap(x, y string) (string, string) {
   return y, x
}

func main() {
   a, b := swap("Mahesh", "Kumar")
   fmt.Println(a, b)
}
```

以上实例执行结果为：

```
Kumar Mahesh
```

函数参数

函数如果使用参数，该变量可称为函数的形参。

形参就像定义在函数体内的局部变量。

调用函数，可以通过两种方式来传递参数：

传递类型	描述
值传递	值传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。
引用传递	引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

函数用法

函数用法	描述
函数作为值	函数定义后可作为值来使用
闭包	闭包是匿名函数，可在动态编程中使用
方法	方法就是一个包含了接受者的函数

Go 语言函数值传递值

传递是指在调用函数时将实际参数复制一份传递到函数中，这样在函数中如果对参数进行修改，将不会影响到实际参数。

默认情况下，Go 语言使用的是值传递，即在调用过程中不会影响到实际参数。

以下定义了 swap() 函数：

```
/* 定义相互交换值的函数 */
func swap(x, y int) int {
   var temp int

   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/

   return temp;
}
```

接下来，让我们使用值传递来调用 swap() 函数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int = 200

   fmt.Printf("交换前 a 的值为 : %d\n", a )
   fmt.Printf("交换前 b 的值为 : %d\n", b )

   /* 通过调用函数来交换值 */
   swap(a, b)

   fmt.Printf("交换后 a 的值 : %d\n", a )
   fmt.Printf("交换后 b 的值 : %d\n", b )
}

/* 定义相互交换值的函数 */
func swap(x, y int) int {
   var temp int

   temp = x /* 保存 x 的值 */
   x = y    /* 将 y 值赋给 x */
   y = temp /* 将 temp 值赋给 y*/

   return temp;
}
```

以下代码执行结果为：

```
交换前 a 的值为 : 100
交换前 b 的值为 : 200
交换后 a 的值 : 100
交换后 b 的值 : 200
```

Go 语言函数引用传递值

引用传递是指在调用函数时将实际参数的地址传递到函数中，那么在函数中对参数所进行的修改，将影响到实际参数。

引用传递指针参数传递到函数内，以下是交换函数 swap() 使用了引用传递：

```
/* 定义交换值函数*/
func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保持 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
}
```

以下我们通过使用引用传递来调用 swap() 函数：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int= 200

   fmt.Printf("交换前，a 的值 : %d\n", a )
   fmt.Printf("交换前，b 的值 : %d\n", b )

   /* 调用 swap() 函数
   * &a 指向 a 指针，a 变量的地址
   * &b 指向 b 指针，b 变量的地址
   */
   swap(&a, &b)

   fmt.Printf("交换后，a 的值 : %d\n", a )
   fmt.Printf("交换后，b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保存 x 地址上的值 */
   *x = *y      /* 将 y 值赋给 x */
   *y = temp    /* 将 temp 值赋给 y */
}
```

以上代码执行结果为：

```
交换前，a 的值 : 100
交换前，b 的值 : 200
交换后，a 的值 : 200
交换后，b 的值 : 100
```

Go 语言函数作为值

Go 语言可以很灵活的创建函数，并作为值使用。以下实例中我们在定义的函数中初始化一个变量，该函数仅仅是为了使用内置函数 math.sqrt()，实例为：

```
package main

import (
   "fmt"
   "math"
)

func main(){
   /* 声明函数变量 */
   getSquareRoot := func(x float64) float64 {
      return math.Sqrt(x)
   }

   /* 使用函数 */
   fmt.Println(getSquareRoot(9))

}
```

以上代码执行结果为：

```
3
```

Go 语言函数闭包

Go 语言支持匿名函数，可作为闭包。匿名函数是一个"内联"语句或表达式。匿名函数的优越性在于可以直接使用函数内的变量，不必申明。

以下实例中，我们创建了函数 getSequence() ，返回另外一个函数。该函数的目的是在闭包中递增 i 变量，代码如下：

```
package main

import "fmt"

func getSequence() func() int {
   i:=0
   return func() int {
      i+=1
     return i
   }
}

func main(){
   /* nextNumber 为一个函数，函数 i 为 0 */
   nextNumber := getSequence()

   /* 调用 nextNumber 函数，i 变量自增 1 并返回 */
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())
   fmt.Println(nextNumber())

   /* 创建新的函数 nextNumber1，并查看结果 */
   nextNumber1 := getSequence()
   fmt.Println(nextNumber1())
   fmt.Println(nextNumber1())
}
```

以上代码执行结果为：

```
1
2
3
1
2
```

Go 语言函数方法

Go 语言中同时有函数和方法。一个方法就是一个包含了接受者的函数，接受者可以是命名类型或者结构体类型的一个值或者是一个指针。所有给定类型的方法属于该类型的方法集。语法格式如下：

```
func (variable_name variable_data_type) function_name() [return_typ
   /* 函数体*/
}
```

下面定义一个结构体类型和该类型的一个方法：

```
package main

import (
   "fmt"
)

/* 定义函数 */
type Circle struct {
  radius float64
}

func main() {
  var c1 Circle
  c1.radius = 10.00
  fmt.Println("Area of Circle(c1) = ", c1.getArea())
}

//该 method 属于 Circle 类型对象中的方法
func (c Circle) getArea() float64 {
  //c.radius 即为 Circle 类型对象中的属性
  return 3.14 * c.radius * c.radius
}
```

以上代码执行结果为：

```
Area of Circle(c1)  =  314
```

Go 语言变量作用域

作用域为已声明标识符所表示的常量、类型、变量、函数或包在源代码中的作用范围。

Go 语言中变量可以在三个地方声明：

- 函数内定义的变量称为局部变量
- 函数外定义的变量称为全局变量
- 函数定义中的变量称为形式参数

接下来让我们具体了解局部变量、全局变量和形式参数。

局部变量

在函数体内声明的变量称之为局部变量，它们的作用域只在函数体内，参数和返回值变量也是局部变量。

以下实例中 main() 函数使用了局部变量 a, b, c：

```
package main

import "fmt"

func main() {
   /* 声明局部变量 */
   var a, b, c int

   /* 初始化参数 */
   a = 10
   b = 20
   c = a + b

   fmt.Printf ("结果： a = %d, b = %d and c = %d\n", a, b, c)
}
```

以上实例执行输出结果为：

```
结果： a = 10, b = 20 and c = 30
```

全局变量

在函数体外声明的变量称之为全局变量，全局变量可以在整个包甚至外部包（被导出后）使用。

全局变量可以在任何函数中使用，以下实例演示了如何使用全局变量：

```
package main

import "fmt"

/* 声明全局变量 */
var g int

func main() {

   /* 声明局部变量 */
   var a, b int

   /* 初始化参数 */
   a = 10
   b = 20
   g = a + b

   fmt.Printf("结果： a = %d, b = %d and g = %d\n", a, b, g)
}
```

以上实例执行输出结果为：

```
结果： a = 10, b = 20 and g = 30
```

Go 语言程序中全局变量与局部变量名称可以相同，但是函数内的局部变量会被优先考虑。实例如下：

```
package main

import "fmt"

/* 声明全局变量 */
var g int = 20

func main() {
   /* 声明局部变量 */
   var g int = 10

   fmt.Printf ("结果： g = %d\n",  g)
}
```

以上实例执行输出结果为：

```
结果： g = 10
```

形式参数

形式参数会作为函数的局部变量来使用。实例如下：

```
package main

import "fmt"

/* 声明全局变量 */
var a int = 20;

func main() {
   /* main 函数中声明局部变量 */
   var a int = 10
   var b int = 20
   var c int = 0

   fmt.Printf("main()函数中 a = %d\n",  a);
   c = sum( a, b);
   fmt.Printf("main()函数中 c = %d\n",  c);
}

/* 函数定义-两数相加 */
func sum(a, b int) int {
   fmt.Printf("sum() 函数中 a = %d\n",  a);
   fmt.Printf("sum() 函数中 b = %d\n",  b);

   return a + b;
}
```

以上实例执行输出结果为：

```
main()函数中 a = 10
sum() 函数中 a = 10
sum() 函数中 b = 20
main()函数中 c = 30
```

初始化局部和全局变量

不同类型的局部和全局变量默认值为：

数据类型	初始化默认值
int	0
float32	0
pointer	nil

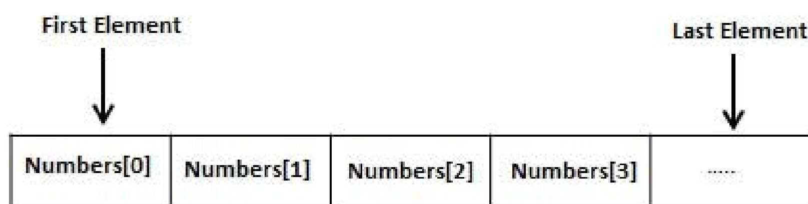
Go 语言数组

Go 语言提供了数组类型的数据结构。

数组是具有相同唯一类型的一组已编号且长度固定的数据项序列，这种类型可以是任意的原始类型例如整形、字符串或者自定义类型。

相对于去声明number0, number1, ..., and number99的变量，使用数组形式numbers[0], numbers[1] ..., numbers[99]更加方便且易于扩展。

数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。



声明数组

Go 语言数组声明需要指定元素类型及元素个数，语法格式如下：

```
var variable_name [SIZE] variable_type
```

以上为一维数组的定义方式。数组长度必须是整数且大于 0。例如以下定义了数组 balance 长度为 10 类型为 float32：

```
var balance [10] float32
```

初始化数组

以下演示了数组初始化：

```
var balance = [5]float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

初始化数组中 {} 中的元素个数不能大于 [] 中的数字。

如果忽略 [] 中的数字不设置数组大小，Go 语言会根据元素的个数来设置数组的大小：

```
var balance = []float32{1000.0, 2.0, 3.4, 7.0, 50.0}
```

该实例与上面的实例是一样的，虽然没有设置数组的大小。

```
balance[4] = 50.0
```

以上实例读取了第五个元素。数组元素可以通过索引（位置）来读取（或者修改），索引从0开始，第一个元素索引为 0，第二个索引为 1，以此类推。

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

访问数组元素

数组元素可以通过索引（位置）来读取。格式为数组名后加中括号，中括号中为索引的值。例如：

```
float32 salary = balance[9]
```

以上实例读取了数组balance第10个元素的值。

以下演示了数组完整操作（声明、赋值、访问）的实例：

```
package main

import "fmt"

func main() {
   var n [10]int /* n 是一个长度为 10 的数组 */
   var i,j int

   /* 为数组 n 初始化元素 */
   for i = 0; i < 10; i++ {
      n[i] = i + 100 /* 设置元素为 i + 100 */
   }

   /* 输出每个数组元素的值 */
   for j = 0; j < 10; j++ {
      fmt.Printf("Element[%d] = %d\n", j, n[j] )
   }
}
```

以上实例执行结果如下：

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

更多内容

数组对 Go 语言来说是非常重要的，以下我们将介绍数组更多的内容：

内容	描述
多维数组	Go 语言支持多维数组，最简单的多维数组是二维数组
向函数传递数组	你可以像函数传递数组参数

Go 语言多维数组

Go 语言支持多维数组，以下为常用的多维数组声明方式：

```
var variable_name [SIZE1][SIZE2]...[SIZEN] variable_type
```

以下实例声明了三维的整型数组：

```
var threedim [5][10][4]int
```

二维数组

二维数组是最简单的多维数组，二维数组本质上是由一维数组组成的。二维数组定义方式如下：

```
var arrayName [ x ][ y ] variable_type
```

variable_type 为 Go 语言的数据类型，arrayName 为数组名，二维数组可认为是一个表格，x 为行，y 为列，下图演示了一个二维数组 a 为三行四列：

	Column 0	Column 1	Column 2	Column 3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]

二维数组中的元素可通过 *a[i][j]* 来访问。

初始化二维数组

多维数组可通过大括号来初始值。以下实例为一个 3 行 4 列的二维数组：

```
a = [3][4]int{
 {0, 1, 2, 3} ,   /*  第一行索引为 0 */
 {4, 5, 6, 7} ,   /*  第二行索引为 1 */
 {8, 9, 10, 11}   /*  第三行索引为 2 */
}
```

访问二维数组

二维数组通过指定坐标来访问。如数组中的行索引与列索引，例如：

```
int val = a[2][3]
```

以上实例访问了二维数组 val 第三行的第四个元素。

二维数组可以使用循环嵌套来输出元素：

```
package main

import "fmt"

func main() {
   /* 数组 - 5 行 2 列*/
   var a = [5][2]int{ {0,0}, {1,2}, {2,4}, {3,6},{4,8}}
   var i, j int

   /* 输出数组元素 */
   for  i = 0; i < 5; i++ {
      for j = 0; j < 2; j++ {
         fmt.Printf("a[%d][%d] = %d\n", i,j, a[i][j] )
      }
   }
}
```

以上实例运行输出结果为：

```
a[0][0] = 0
a[0][1] = 0
a[1][0] = 1
a[1][1] = 2
a[2][0] = 2
a[2][1] = 4
a[3][0] = 3
a[3][1] = 6
a[4][0] = 4
a[4][1] = 8
```

Go 语言向函数传递数组

如果你想向函数传递数组参数，你需要在函数定义时，声明形参为数组，我们可以通过以下两种方式来声明：

方式一

形参设定数组大小：

```
void myFunction(param [10]int)
{
.
.
.
}
```

方式二

形参未设定数组大小：

```
void myFunction(param []int)
{
.
.
.
}
```

实例

让我们看下以下实例，实例中函数接收整型数组参数，另一个参数指定了数组元素的个数，并返回平均值：

```
func getAverage(arr []int, int size) float32
{
   var i int
   var avg, sum float32

   for i = 0; i < size; ++i {
      sum += arr[i]
   }

   avg = sum / size

   return avg;
}
```

接下来我们来调用这个函数：

```
package main

import "fmt"

func main() {
   /* 数组长度为 5 */
   var  balance = []int {1000, 2, 3, 17, 50}
   var avg float32

   /* 数组作为参数传递给函数 */
   avg = getAverage( balance, 5 ) ;

   /* 输出返回的平均值 */
   fmt.Printf( "平均值为: %f ", avg );
}
func getAverage(arr []int, size int) float32 {
   var i,sum int
   var avg float32

   for i = 0; i < size;i++ {
      sum += arr[i]
   }

   avg = float32(sum / size)

   return avg;
}
```

以上实例执行输出结果为：

```
平均值为:  214.000000
```

以上实例中我们使用的形参并为设定数组大小。

Go 语言指针

Go 语言中指针是很容易学习的，Go 语言中使用指针可以更简单的执行一些任务。

接下来让我们来一步步学习 Go 语言指针。

我们都知道，变量是一种使用方便的占位符，用于引用计算机内存地址。

Go 语言的取地址符是 &，放到一个变量前使用就会返回相应变量的内存地址。

以下实例演示了变量在内存中地址：

```
package main

import "fmt"

func main() {
   var a int = 10

   fmt.Printf("变量的地址: %x\n", &a  )
}
```

执行以上代码输出结果为：

```
变量的地址: 20818a220
```

现在我们已经了解了什么是内存地址和如何去反问它。接下来我们将具体介绍指针。

什么是指针

一个指针变量可以指向任何一个值的内存地址它指向那个值的内存地址。

类似于变量和常量，在使用指针前你需要声明指针。指针声明格式如下：

```
var var_name *var-type
```

var-type 为指针类型，var_name 为指针变量名，* 号用于指定变量是作为一个指针。以下是有效的指针声明：

```
var ip *int        /* 指向整型*/
var fp *float32    /* 指向浮点型 */
```

本例中这是一个指向 int 和 float32 的指针。

如何使用指针

指针使用流程：

- 定义指针变量。
- 为指针变量赋值。
- 访问指针变量中指向地址的值。

在指针类型前面加上 * 号（前缀）来获取指针所指向的内容。

```
package main

import "fmt"

func main() {
   var a int= 20   /* 声明实际变量 */
   var ip *int        /* 声明指针变量 */

   ip = &a  /* 指针变量的存储地址 */

   fmt.Printf("a 变量的地址是: %x\n", &a  )

   /* 指针变量的存储地址 */
   fmt.Printf("ip 变量的存储地址: %x\n", ip )

   /* 使用指针访问值 */
   fmt.Printf("*ip 变量的值: %d\n", *ip )
}
```

以上实例执行输出结果为：

```
a 变量的地址是: 20818a220
ip 变量的存储地址: 20818a220
*ip 变量的值: 20
```

Go 空指针

当一个指针被定义后没有分配到任何变量时，它的值为 nil。

nil 指针也称为空指针。

nil在概念上和其它语言的null、None、nil、NULL一样，都指代零值或空值。

一个指针变量通常缩写为 ptr。

查看以下实例：

```
package main

import "fmt"

func main() {
   var  ptr *int

   fmt.Printf("ptr 的值为 : %x\n", ptr  )
}
```

以上实例输出结果为：

```
ptr 的值为 : 0
```

空指针判断：

```
if(ptr != nil)     /* ptr 不是空指针 */
if(ptr == nil)    /* ptr 是空指针 */
```

Go指针更多内容

接下来我们将为大家介绍Go语言中更多的指针应用：

内容	描述
Go 指针数组	你可以定义一个指针数组来存储地址
Go 指向指针的指针	Go 支持指向指针的指针
Go 像函数传递指针参数	通过引用或地址传参，在函数调用时可以改变其值

Go 语言指针数组

在我们了解指针数组前，先看个实例，定义了长度为 3 的整型数组：

```
package main

import "fmt"

const MAX int = 3

func main() {

   a := []int{10,100,200}
   var i int

   for i = 0; i < MAX; i++ {
      fmt.Printf("a[%d] = %d\n", i, a[i] )
   }
}
```

以上代码执行输出结果为：

```
a[0] = 10
a[1] = 100
a[2] = 200
```

有一种情况，我们可能需要保存数组，这样我们就需要使用到指针。

以下声明了整型指针数组：

```
var ptr [MAX]*int;
```

ptr 为整型指针数组。因此每个元素都指向了一个值。以下实例的三个整数将存储在指针数组中：

```
package main

import "fmt"

const MAX int = 3

func main() {
   a := []int{10,100,200}
   var i int
   var ptr [MAX]*int;

   for  i = 0; i < MAX; i++ {
      ptr[i] = &a[i] /* 整数地址赋值给指针数组 */
   }

   for  i = 0; i < MAX; i++ {
      fmt.Printf("a[%d] = %d\n", i,*ptr[i] )
   }
}
```

以上代码执行输出结果为：

```
a[0] = 10
a[1] = 100
a[2] = 200
```

Go 语言指向指针的指针

如果一个指针变量存放的又是另一个指针变量的地址，则称这个指针变量为指向指针的指针变量。

当定义一个指向指针的指针变量时，第一个指针存放第二个指针的地址，第二个指针存放变量的地址：



指向指针的指针变量声明格式如下：

```
var ptr **int;
```

以上指向指针的指针变量为整型。

访问指向指针的指针变量值需要使用两个 * 号，如下所示：

```
package main

import "fmt"

func main() {

   var a int
   var ptr *int
   var pptr **int

   a = 3000

   /* 指针 ptr 地址 */
   ptr = &a

   /* 指向指针 ptr 地址 */
   pptr = &ptr

   /* 获取 pptr 的值 */
   fmt.Printf("变量 a = %d\n", a )
   fmt.Printf("指针变量 *ptr = %d\n", *ptr )
   fmt.Printf("指向指针的指针变量 **pptr = %d\n", **pptr)
}
```

以上实例执行输出结果为：

```
变量 a = 3000
指针变量 *ptr = 3000
指向指针的指针变量 **pptr = 3000
```

Go 语言指针作为函数参数

Go 语言允许向函数传递指针，志需要在函数定义的参数上设置为指针类型即可。

以下实例演示了如何向函数传递指针，并在函数调用后修改函数内的值，：

```
package main

import "fmt"

func main() {
   /* 定义局部变量 */
   var a int = 100
   var b int= 200

   fmt.Printf("交换前 a 的值 : %d\n", a )
   fmt.Printf("交换前 b 的值 : %d\n", b )

   /* 调用函数用于交换值
   * &a 指向 a 变量的地址
   * &b 指向 b 变量的地址
   */
   swap(&a, &b);

   fmt.Printf("交换后 a 的值 : %d\n", a )
   fmt.Printf("交换后 b 的值 : %d\n", b )
}

func swap(x *int, y *int) {
   var temp int
   temp = *x    /* 保存 x 地址的值 */
   *x = *y      /* 将 y 赋值给 x */
   *y = temp    /* 将 temp 赋值给 y */
}
```

以上实例允许输出结果为：

```
交换前 a 的值 : 100
交换前 b 的值 : 200
交换后 a 的值 : 200
交换后 b 的值 : 100
```

Go 语言结构体

Go 语言中数组可以存储同一类型的数据，但在结构体中我们可以为不同项定义不同的数据类型。

结构体是由一系列具有相同类型或不同类型的数据构成的数据集合。

结构体表示一项记录，比如保存图书馆的书籍记录，每本书有以下属性：

- Title ：标题
- Author ： 作者
- Subject：学科
- ID：书籍ID

定义结构体

结构体定义需要使用 type 和 struct 语句。struct 语句定义一个新的数据类型，结构体有中一个或多个成员。type 语句设定了结构体的名称。结构体的格式如下：

```
type struct_variable_type struct {
   member definition;
   member definition;
   ...
   member definition;
}
```

一旦定义了结构体类型，它就能用于变量的声明，语法格式如下：

```
variable_name := structure_variable_type {value1, value2...valuen}
```

访问结构体成员

如果要访问结构体成员，需要使用点号 (.) 操作符，格式为："结构体.成员名"。

结构体类型变量使用struct关键字定义，实例如下：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   fmt.Printf( "Book 1 title : %s\n", Book1.title)
   fmt.Printf( "Book 1 author : %s\n", Book1.author)
   fmt.Printf( "Book 1 subject : %s\n", Book1.subject)
   fmt.Printf( "Book 1 book_id : %d\n", Book1.book_id)

   /* 打印 Book2 信息 */
   fmt.Printf( "Book 2 title : %s\n", Book2.title)
   fmt.Printf( "Book 2 author : %s\n", Book2.author)
   fmt.Printf( "Book 2 subject : %s\n", Book2.subject)
   fmt.Printf( "Book 2 book_id : %d\n", Book2.book_id)
}
```

以上实例执行运行结果为：

```
Book 1 title : Go 语言
Book 1 author : www.runoob.com
Book 1 subject : Go 语言教程
Book 1 book_id : 6495407
Book 2 title : Python 教程
Book 2 author : www.runoob.com
Book 2 subject : Python 语言教程
Book 2 book_id : 6495700
```

结构体作为函数参数

你可以向其他数据类型一样将结构体类型作为参数传递给函数。并以以上实例的方式访问结构体变量：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* 声明 Book1 为 Books 类型 */
   var Book2 Books        /* 声明 Book2 为 Books 类型 */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(Book1)

   /* 打印 Book2 信息 */
   printBook(Book2)
}

func printBook( book Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

以上实例执行运行结果为：

```
Book title : Go 语言
Book author : www.runoob.com
Book subject : Go 语言教程
Book book_id : 6495407
Book title : Python 教程
Book author : www.runoob.com
Book subject : Python 语言教程
Book book_id : 6495700
```

结构体指针

你可以定义指向结构体的指针类似于其他指针变量，格式如下：

```
var struct_pointer *Books
```

以上定义的指针变量可以存储结构体变量的地址。查看结构体变量地址，可以将 & 符号放置于结构体变量前：

```
struct_pointer = &Book1;
```

使用结构体指针访问结构体成员，使用 "." 操作符：

```
struct_pointer.title;
```

接下来让我们使用结构体指针重写以上实例，代码如下：

```
package main

import "fmt"

type Books struct {
   title string
   author string
   subject string
   book_id int
}

func main() {
   var Book1 Books        /* Declare Book1 of type Book */
   var Book2 Books        /* Declare Book2 of type Book */

   /* book 1 描述 */
   Book1.title = "Go 语言"
   Book1.author = "www.runoob.com"
   Book1.subject = "Go 语言教程"
   Book1.book_id = 6495407

   /* book 2 描述 */
   Book2.title = "Python 教程"
   Book2.author = "www.runoob.com"
   Book2.subject = "Python 语言教程"
   Book2.book_id = 6495700

   /* 打印 Book1 信息 */
   printBook(&Book1)

   /* 打印 Book2 信息 */
   printBook(&Book2)
}
func printBook( book *Books ) {
   fmt.Printf( "Book title : %s\n", book.title);
   fmt.Printf( "Book author : %s\n", book.author);
   fmt.Printf( "Book subject : %s\n", book.subject);
   fmt.Printf( "Book book_id : %d\n", book.book_id);
}
```

以上实例执行运行结果为：

```
Book title : Go 语言
Book author : www.runoob.com
Book subject : Go 语言教程
Book book_id : 6495407
Book title : Python 教程
Book author : www.runoob.com
Book subject : Python 语言教程
Book book_id : 6495700
```

Go 语言切片(Slice)

Go 语言切片是对数组的抽象。

Go 数组的长度不可改变，在特定场景中这样的集合就不太适用，Go中提供了一种灵活，功能强悍的内置类型切片("动态数组"),与数组相比切片的长度是不固定的，可以追加元素，在追加时可能使切片的容量增大。

定义切片

你可以声明一个未指定大小的数组来定义切片：

```
var identifier []type
```

切片不需要说明长度。

或使用make()函数来创建切片:

```
var slice1 []type = make([]type, len)

也可以简写为

slice1 := make([]type, len)
```

也可以指定容量，其中capacity为可选参数。

```
make([]T, length, capacity)
```

这里 len 是数组的长度并且也是切片的初始长度。

切片初始化

```
s :=[] int {1,2,3 }
```

直接初始化切片，[]表示是切片类型，{1,2,3}初始化值依次是1,2,3.其cap=len=3

```
s := arr[:]
```

初始化切片s,是数组arr的引用

```
s := arr[startIndex:endIndex]
```

将arr中从下标startIndex到endIndex-1 下的元素创建为一个新的切片

```
s := arr[startIndex:]
```

缺省endIndex时将表示一直到arr的最后一个元素

```
s := arr[:endIndex]
```

缺省startIndex时将表示从arr的第一个元素开始

```
s1 := s[startIndex:endIndex]
```

通过切片s初始化切片s1

```
s :=make([]int,len,cap)
```

通过内置函数make()初始化切片s,[]int 标识为其元素类型为int的切片

len() 和 cap() 函数

切片是可索引的，并且可以由 len() 方法获取长度。

切片提供了计算容量的方法 cap() 可以测量切片最长可以达到多少。

以下为具体实例：

```
package main

import "fmt"

func main() {
   var numbers = make([]int,3,5)

   printSlice(numbers)
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

以上实例运行输出结果为:

```
len=3 cap=5 slice=[0 0 0]
```

空(nil)切片

一个切片在未初始化之前默认为 nil，长度为 0，实例如下：

```
package main

import "fmt"

func main() {
   var numbers []int

   printSlice(numbers)

   if(numbers == nil){
      fmt.Printf("切片是空的")
   }
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

以上实例运行输出结果为:

```
len=0 cap=0 slice=[]
切片是空的
```

切片截取

可以通过设置下限及上限来设置截取切片 *[lower-bound:upper-bound]*，实例如下：

```
package main

import "fmt"

func main() {
   /* 创建切片 */
   numbers := []int{0,1,2,3,4,5,6,7,8}
   printSlice(numbers)

   /* 打印原始切片 */
   fmt.Println("numbers ==", numbers)

   /* 打印子切片从索引1(包含) 到索引4(不包含)*/
   fmt.Println("numbers[1:4] ==", numbers[1:4])

   /* 默认下限为 0*/
   fmt.Println("numbers[:3] ==", numbers[:3])

   /* 默认上限为 len(s)*/
   fmt.Println("numbers[4:] ==", numbers[4:])

   numbers1 := make([]int,0,5)
   printSlice(numbers1)

   /* 打印子切片从索引  0(包含) 到索引 2(不包含) */
   number2 := numbers[:2]
   printSlice(number2)

   /* 打印子切片从索引 2(包含) 到索引 5(不包含) */
   number3 := numbers[2:5]
   printSlice(number3)

}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

执行以上代码输出结果为：

```
len=9 cap=9 slice=[0 1 2 3 4 5 6 7 8]
numbers == [0 1 2 3 4 5 6 7 8]
numbers[1:4] == [1 2 3]
numbers[:3] == [0 1 2]
numbers[4:] == [4 5 6 7 8]
len=0 cap=5 slice=[]
len=2 cap=9 slice=[0 1]
len=3 cap=7 slice=[2 3 4]
```

append() 和 copy() 函数

如果想增加切片的容量，我们必须创建一个新的更大的切片并把原分片的内容都拷贝过来。

下面的代码描述了从拷贝切片的 copy 方法和向切片追加新元素的 append 方法。

```
package main

import "fmt"

func main() {
   var numbers []int
   printSlice(numbers)

   /* 允许追加空切片 */
   numbers = append(numbers, 0)
   printSlice(numbers)

   /* 向切片添加一个元素 */
   numbers = append(numbers, 1)
   printSlice(numbers)

   /* 同时添加多个元素 */
   numbers = append(numbers, 2,3,4)
   printSlice(numbers)

   /* 创建切片 numbers1 是之前切片的两倍容量*/
   numbers1 := make([]int, len(numbers), (cap(numbers))*2)

   /* 拷贝 numbers 的内容到 numbers1 */
   copy(numbers1,numbers)
   printSlice(numbers1)
}

func printSlice(x []int){
   fmt.Printf("len=%d cap=%d slice=%v\n",len(x),cap(x),x)
}
```

以上代码执行输出结果为：

```
len=0 cap=0 slice=[]
len=1 cap=2 slice=[0]
len=2 cap=2 slice=[0 1]
len=5 cap=8 slice=[0 1 2 3 4]
len=5 cap=16 slice=[0 1 2 3 4]
```

Go 语言范围(Range)

Go 语言中 range 关键字用于for循环中迭代数组(array)、切片(slice)、链表(channel)或集合(map)的元素。在数组和切片中它返回元素的索引值，在集合中返回 key-value 对的 key 值。

实例

```
package main
import "fmt"
func main() {
    //这是我们使用range去求一个slice的和。使用数组跟这个很类似
    nums := []int{2, 3, 4}
    sum := 0
    for _, num := range nums {
        sum += num
    }
    fmt.Println("sum:", sum)
    //在数组上使用range将传入index和值两个变量。上面那个例子我们不需要使用该
    for i, num := range nums {
        if num == 3 {
            fmt.Println("index:", i)
        }
    }
    //range也可以用在map的键值对上。
    kvs := map[string]string{"a": "apple", "b": "banana"}
    for k, v := range kvs {
        fmt.Printf("%s -> %s\n", k, v)
    }
    //range也可以用来枚举Unicode字符串。第一个参数是字符的索引，第二个是字符
    for i, c := range "go" {
        fmt.Println(i, c)
    }
}
```

以上实例运行输出结果为：

```
sum: 9
index: 1
a -> apple
b -> banana
0 103
1 111
```

Go 语言Map(集合)

Map 是一种无序的键值对的集合。Map 最重要的一点是通过 key 来快速检索数据，key 类似于索引，指向数据的值。

Map 是一种集合，所以我们可以像迭代数组和切片那样迭代它。不过，Map 是无序的，我们无法决定它的返回顺序，这是因为 Map 是使用 hash 表来实现的。

定义 Map

可以使用内建函数 make 也可以使用 map 关键字来定义 Map:

```
/* 声明变量，默认 map 是 nil */
var map_variable map[key_data_type]value_data_type

/* 使用 make 函数 */
map_variable = make(map[key_data_type]value_data_type)
```

如果不初始化 map，那么就会创建一个 nil map。nil map 不能用来存放键值对

实例

下面实例演示了创建和使用map:

```
package main

import "fmt"

func main() {
   var countryCapitalMap map[string]string
   /* 创建集合 */
   countryCapitalMap = make(map[string]string)

   /* map 插入 key-value 对，各个国家对应的首都 */
   countryCapitalMap["France"] = "Paris"
   countryCapitalMap["Italy"] = "Rome"
   countryCapitalMap["Japan"] = "Tokyo"
   countryCapitalMap["India"] = "New Delhi"

   /* 使用 key 输出 map 值 */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[count
   }

   /* 查看元素在集合中是否存在 */
   captial, ok := countryCapitalMap["United States"]
   /* 如果 ok 是 true, 则存在，否则不存在 */
   if(ok){
      fmt.Println("Capital of United States is", captial)
   }else {
      fmt.Println("Capital of United States is not present")
   }
}
```

以上实例运行结果为：

```
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Capital of United States is not present
```

delete() 函数

delete() 函数用于删除集合的元素, 参数为 map 和其对应的 key。实例如下：

```
package main

import "fmt"

func main() {
   /* 创建 map */
   countryCapitalMap := map[string] string {"France":"Paris","Italy

   fmt.Println("原始 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[count
   }

   /* 删除元素 */
   delete(countryCapitalMap,"France");
   fmt.Println("Entry for France is deleted")

   fmt.Println("删除元素后 map")

   /* 打印 map */
   for country := range countryCapitalMap {
      fmt.Println("Capital of",country,"is",countryCapitalMap[count
   }
}
```

以上实例运行结果为：

```
原始 map
Capital of France is Paris
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
Entry for France is deleted
删除元素后 map
Capital of Italy is Rome
Capital of Japan is Tokyo
Capital of India is New Delhi
```

Go 语言递归函数

递归，就是在运行的过程中调用自己。

语法格式如下：

```
func recursion() {
   recursion() /* 函数调用自身 */
}

func main() {
   recursion()
}
```

Go 语言支持递归。但我们在使用递归时，开发者需要设置退出条件，否则递归将陷入无限循环中。

递归函数对于解决数学上的问题是非常有用的，就像计算阶乘，生成斐波那契数列等。

阶乘

以下实例通过 Go 语言的递归函数实例阶乘：

```
package main

import "fmt"

func Factorial(x int) (result int) {
  if x == 0 {
    result = 1;
  } else {
    result = x * Factorial(x - 1);
  }
  return;
}

func main() {
    var i int = 15
    fmt.Printf("%d 的阶乘是 %d\n", i, Factorial(i))
}
```

以上实例执行输出结果为：

```
15 的阶乘是 1307674368000
```

斐波那契数列

以下实例通过 Go 语言的递归函数实现斐波那契数列：

```
package main

import "fmt"

func fibonaci(n int) int {
  if n < 2 {
   return n
  }
  return fibonaci(n-2) + fibonaci(n-1)
}

func main() {
    var i int
    for i = 0; i < 10; i++ {
       fmt.Printf("%d\t", fibonaci(i))
    }
}
```

以上实例执行输出结果为：

```
0    1    1    2    3    5    8    13    21    34
```

Go 语言类型转换

类型转换用于将一种数据类型的变量转换为另外一种类型的变量。Go 语言类型转换基本格式如下：

```
type_name(expression)
```

type_name 为类型，expression 为表达式。

实例

以下实例中将整型转化为浮点型，并计算结果，将结果赋值给浮点型变量：

```
package main

import "fmt"

func main() {
   var sum int = 17
   var count int = 5
   var mean float32

   mean = float32(sum)/float32(count)
   fmt.Printf("mean 的值为: %f\n",mean)
}
```

以上实例执行输出结果为：

```
mean 的值为: 3.400000
```

Go 语言接口

Go 语言提供了另外一种数据类型即接口，它把所有的具有共性的方法定义在一起，任何其他类型只要实现了这些方法就是实现了这个接口。

实例

```
/* 定义接口 */
type interface_name interface {
   method_name1 [return_type]
   method_name2 [return_type]
   method_name3 [return_type]
   ...
   method_namen [return_type]
}

/* 定义结构体 */
type struct_name struct {
   /* variables */
}

/* 实现接口方法 */
func (struct_name_variable struct_name) method_name1() [return_type
   /* 方法实现 */
}
...
func (struct_name_variable struct_name) method_namen() [return_type
   /* 方法实现*/
}
```

实例

```
package main

import (
    "fmt"
)

type Phone interface {
    call()
}

type NokiaPhone struct {
}

func (nokiaPhone NokiaPhone) call() {
    fmt.Println("I am Nokia, I can call you!")
}

type IPhone struct {
}

func (iPhone IPhone) call() {
    fmt.Println("I am iPhone, I can call you!")
}

func main() {
    var phone Phone

    phone = new(NokiaPhone)
    phone.call()

    phone = new(IPhone)
    phone.call()

}
```

在上面的例子中，我们定义了一个接口Phone，接口里面有一个方法call()。然后我们在main函数里面定义了一个Phone类型变量，并分别为之赋值为NokiaPhone和IPhone。然后调用call()方法，输出结果如下：

```
I am Nokia, I can call you!
I am iPhone, I can call you!
```

Go 错误处理

Go 语言通过内置的错误接口提供了非常简单的错误处理机制。

error类型是一个接口类型，这是它的定义：

```
type error interface {
    Error() string
}
```

我们可以在编码中通过实现 error 接口类型来生成错误信息。

函数通常在最后的返回值中返回错误信息。使用errors.New 可返回一个错误信息：

```
func Sqrt(f float64) (float64, error) {
    if f < 0 {
        return 0, errors.New("math: square root of negative number'
    }
    // 实现
}
```

在下面的例子中，我们在调用Sqrt的时候传递的一个负数，然后就得到了non-nil的error对象，将此对象与nil比较，结果为true，所以fmt.Println(fmt包在处理error时会调用Error方法)被调用，以输出错误，请看下面调用的示例代码：

```
result, err:= Sqrt(-1)

if err != nil {
   fmt.Println(err)
}
```

实例

```
package main

import (
    "fmt"
)

// 定义一个 DivideError 结构
type DivideError struct {
    dividee int
    divider int
}

// 实现      `error` 接口
func (de *DivideError) Error() string {
    strFormat := `
    Cannot proceed, the divider is zero.
    dividee: %d
    divider: 0
`
    return fmt.Sprintf(strFormat, de.dividee)
}

// 定义 `int` 类型除法运算的函数
func Divide(varDividee int, varDivider int) (result int, errorMsg s
    if varDivider == 0 {
        dData := DivideError{
            dividee: varDividee,
            divider: varDivider,
        }
        errorMsg = dData.Error()
        return
    } else {
        return varDividee / varDivider, ""
    }

}

func main() {

    // 正常情况
    if result, errorMsg := Divide(100, 10); errorMsg == "" {
        fmt.Println("100/10 = ", result)
    }
    // 当被除数为零的时候会返回错误信息
    if _, errorMsg := Divide(100, 0); errorMsg != "" {
        fmt.Println("errorMsg is: ", errorMsg)
    }

}
```

执行以上程序，输出结果为：

```
100/10 =  10
errorMsg is: 
    Cannot proceed, the divider is zero.
    dividee: 100
    divider: 0
```

Go 语言开发工具

LiteIDE

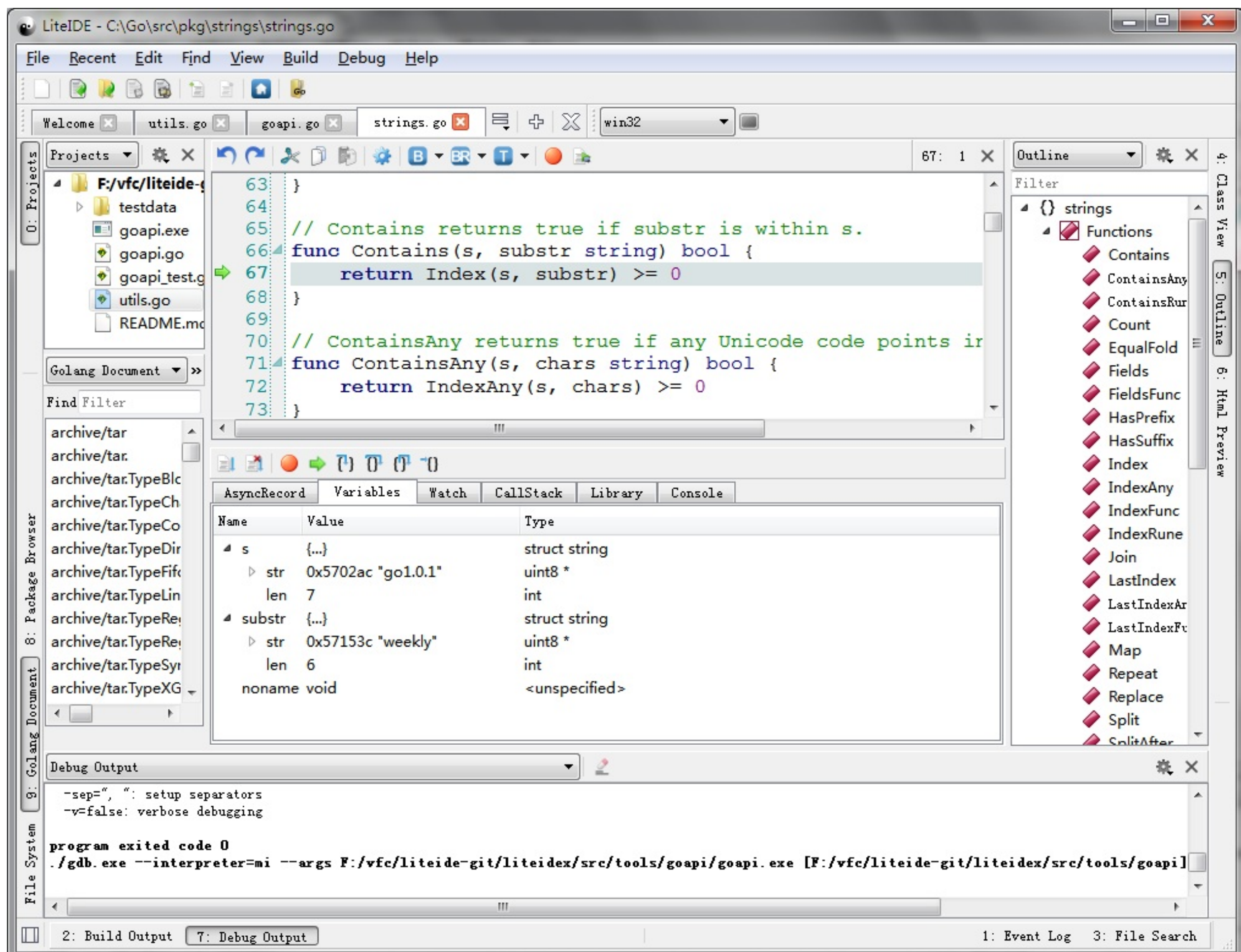
LiteIDE是一款开源、跨平台的轻量级Go语言集成开发环境（IDE）。

支持的操作系统

- Windows x86 (32-bit or 64-bit)
- Linux x86 (32-bit or 64-bit)

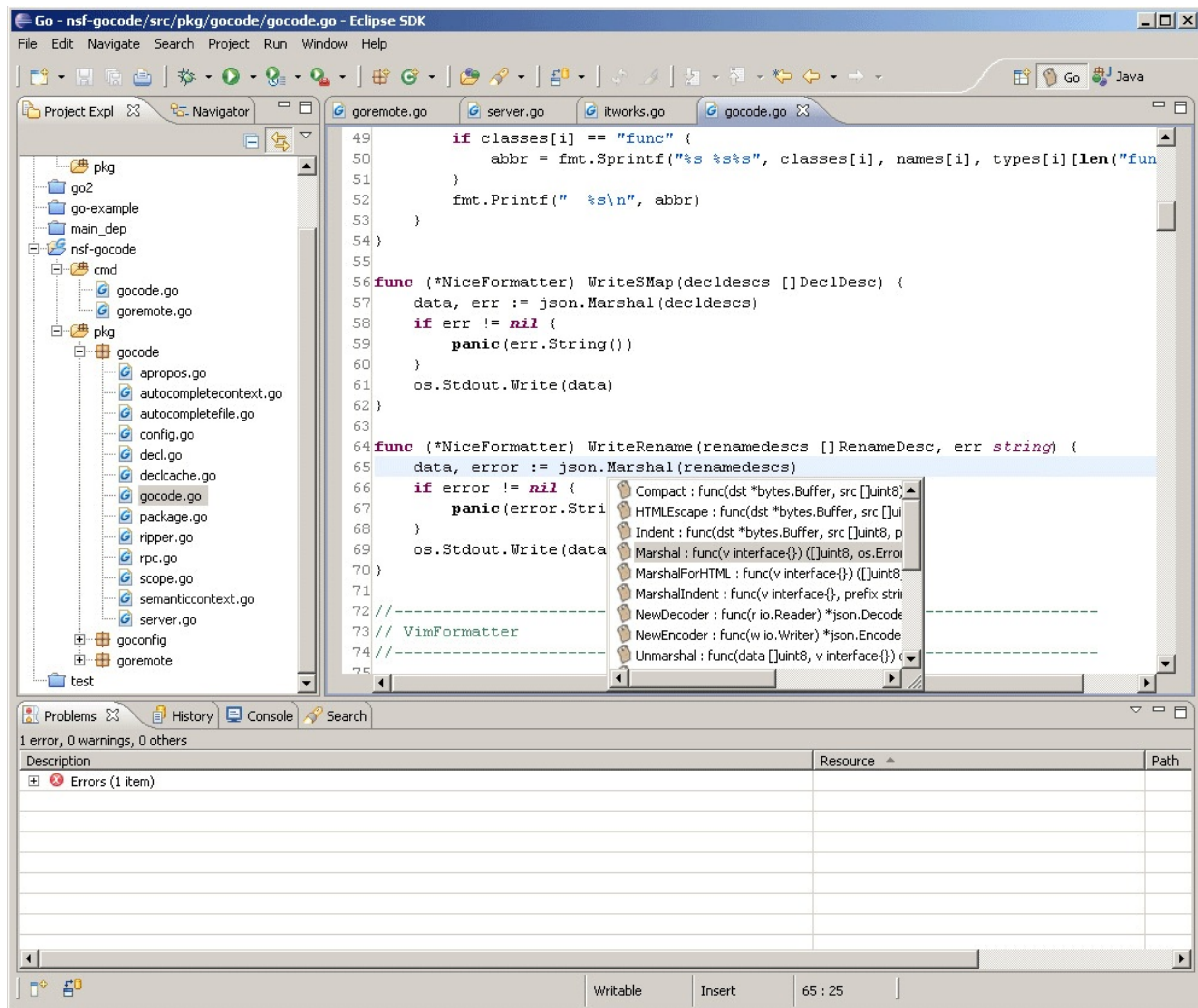
下载地址 ：http://sourceforge.net/projects/liteide/files/

源码地址 ：https://github.com/visualfc/liteide



Eclipse

Eclipse也是非常常用的开发利器，以下介绍如何使用Eclipse来编写Go程序。



Eclipse编辑Go的主界面

1. 首先下载并安装好Eclipse
2. 下载goclipse插件
   http://code.google.com/p/goclipse/wiki/InstallationInstructions
3. 下载gocode，用于go的代码补全提示

   gocode的github地址：

   ```
   https://github.com/nsf/gocode
   ```

   在windows下要安装git，通常用msysgit

   再在cmd下安装：

   ```
   go get -u github.com/nsf/gocode
   ```

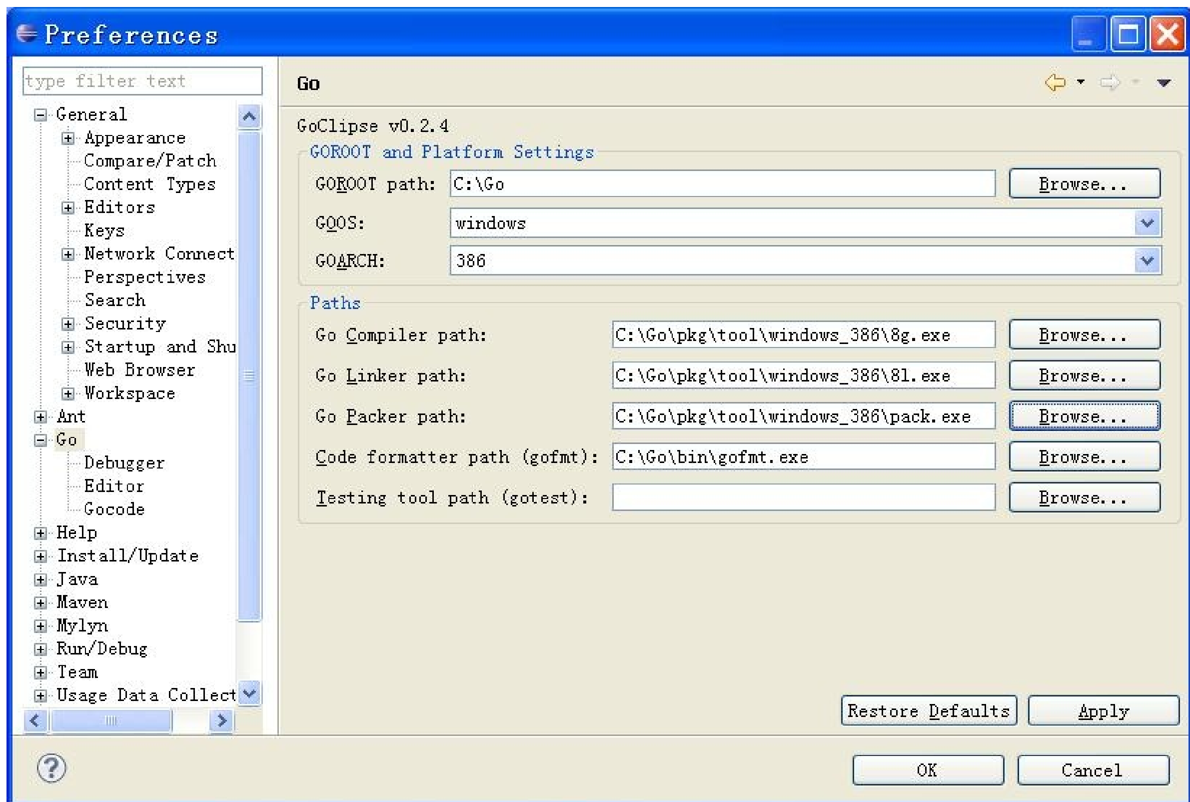
   也可以下载代码，直接用go build来编译，会生成gocode.exe

4. 下载MinGW并按要求装好

5. 配置插件

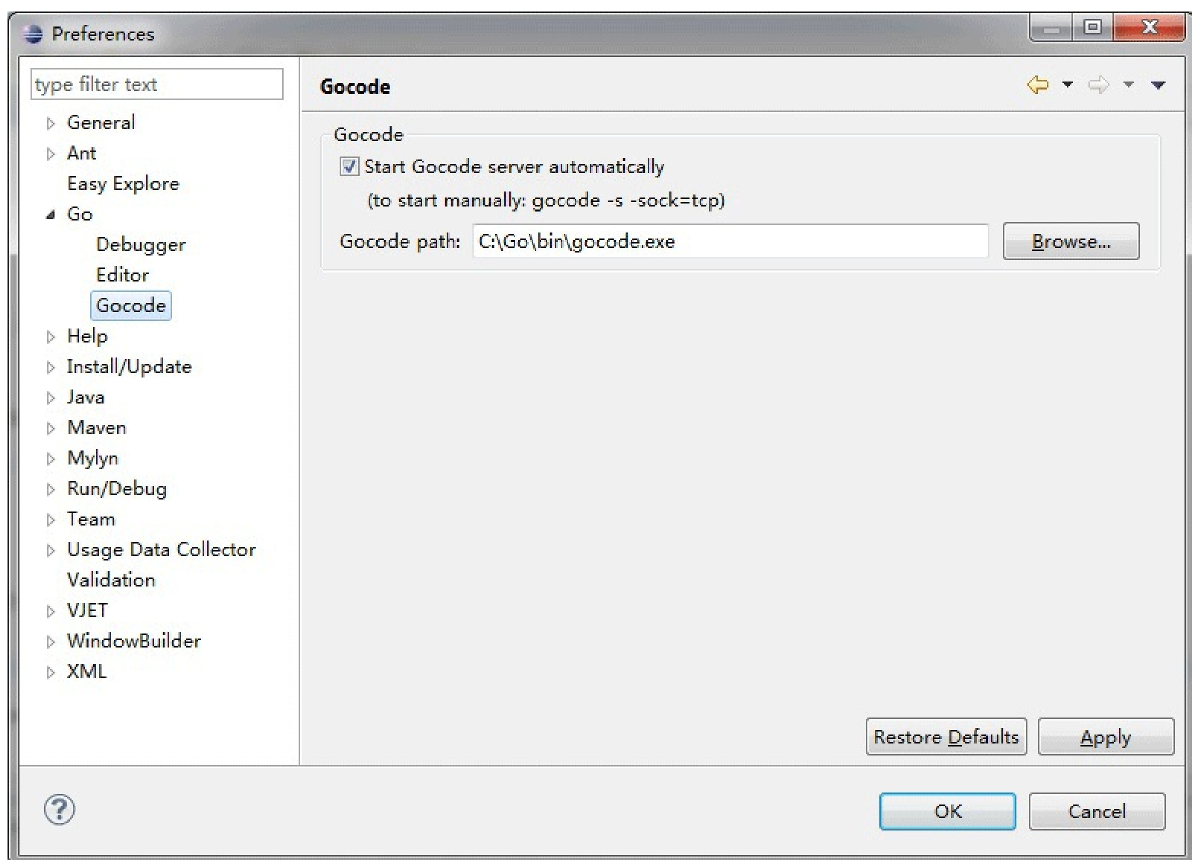
Windows->Reference->Go

(1).配置Go的编译器



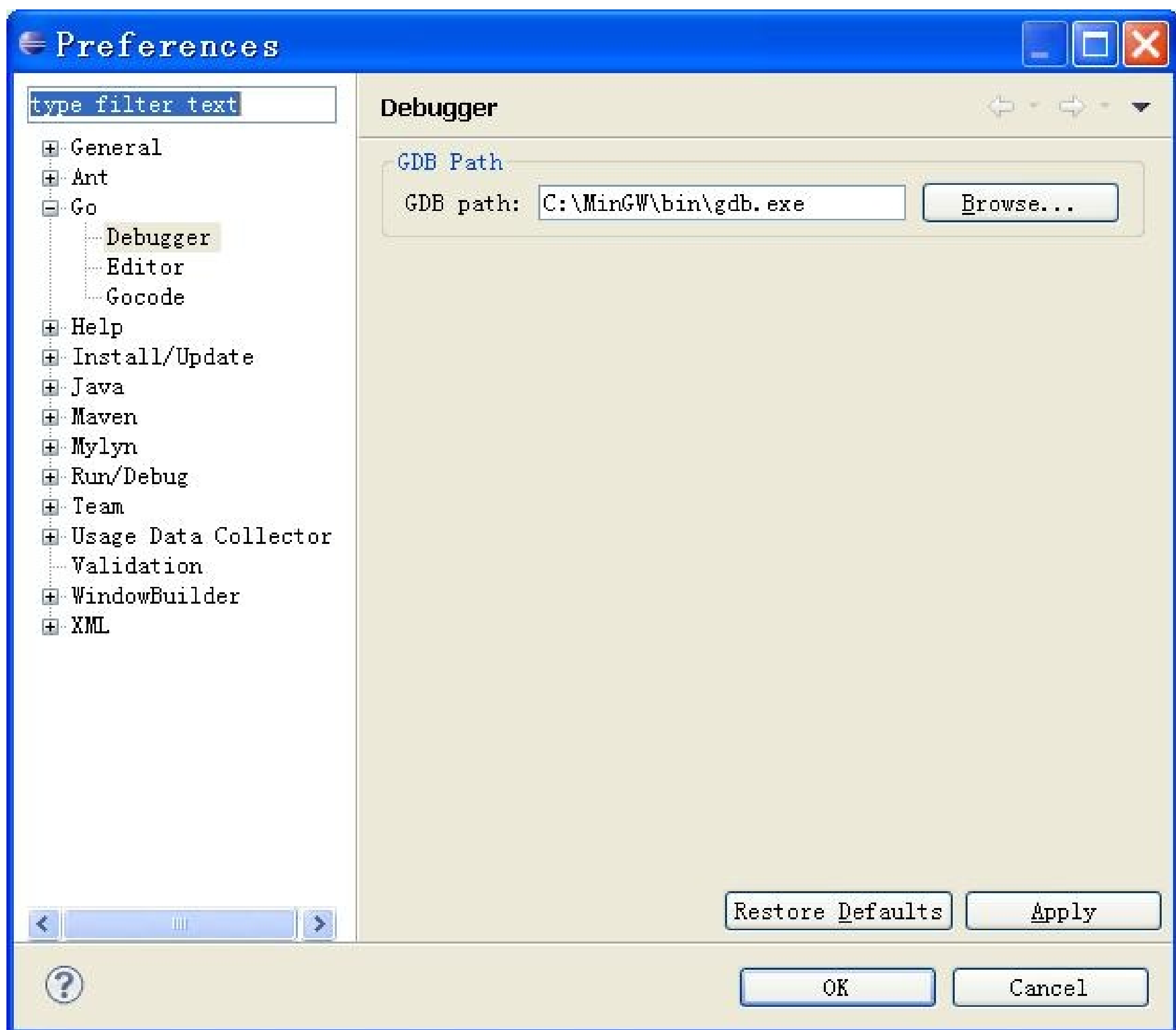
设置Go的一些基础信息

(2).配置Gocode（可选，代码补全），设置Gocode路径为之前生成的gocode.exe文件



设置gocode信息

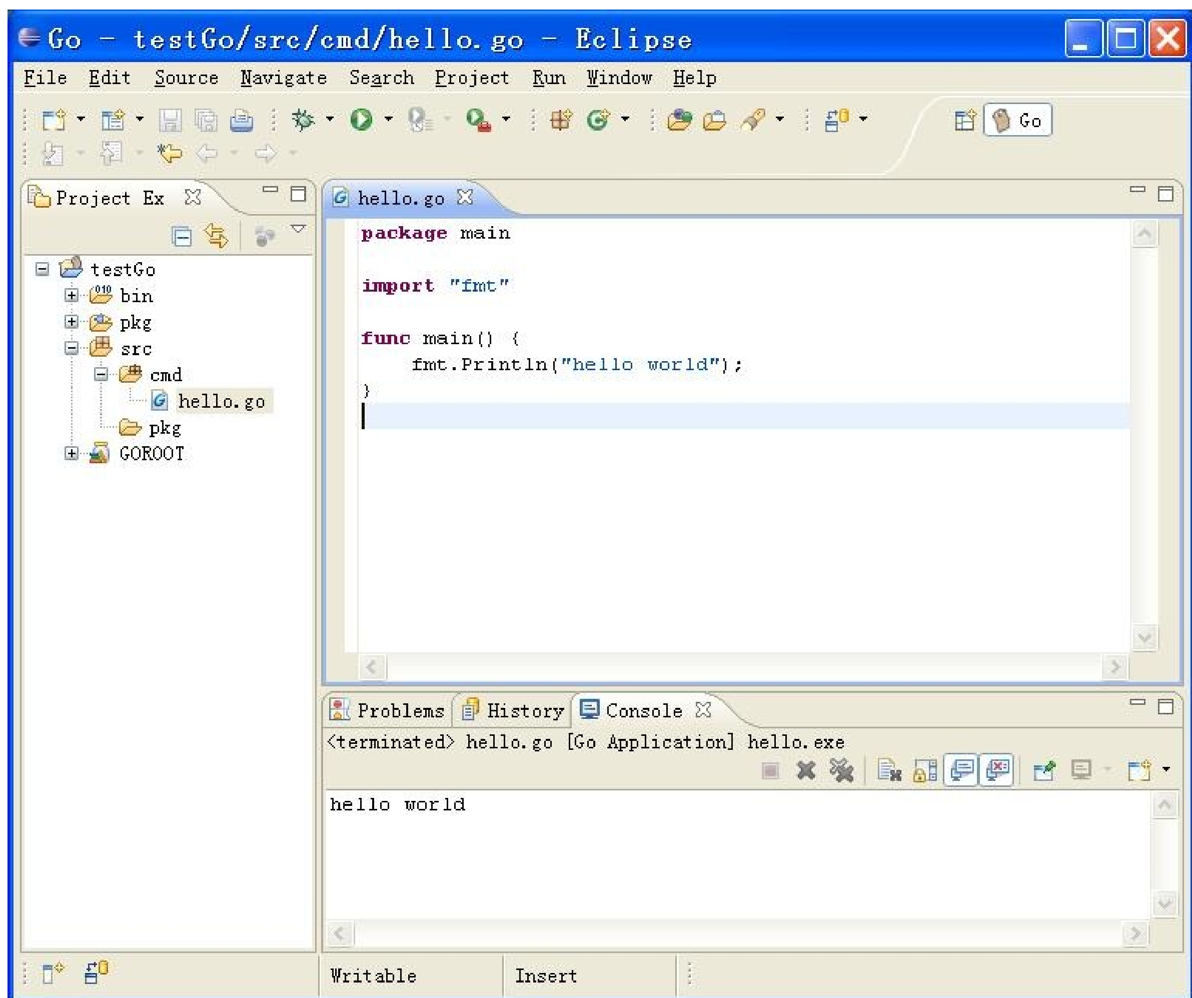
(3).配置GDB（可选，做调试用），设置GDB路径为MingW安装目录下的gdb.exe文件



设置GDB信息

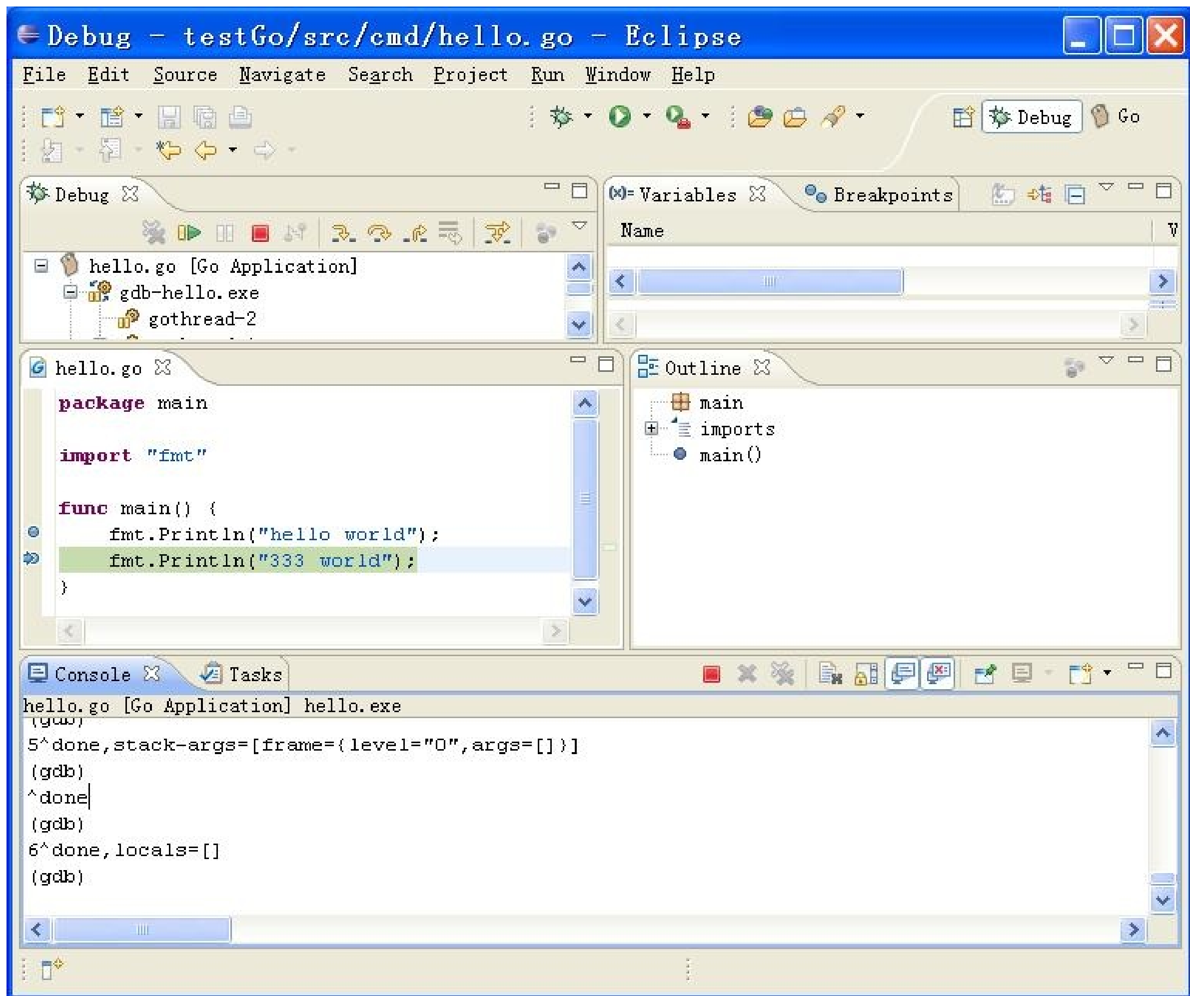
6. 测试是否成功

新建一个go工程，再建立一个hello.go。如下图：



新建项目编辑文件

调试如下（要在console中用输入命令来调试）：


Debug - testGo/src/cmd/hello.go - Eclipse
File Edit Source Navigate Search Project Run Window Help
Debug
Go
Debug
hello.go [Go Application]
gdb-hello.exe
gothread-2
Variables
Breakpoints
Name
hello.go
package main
import "fmt"
func main() {
fmt.Println("hello world");
fmt.Println("333 world");
}
Outline
main
imports
main()
Console
Tasks
hello.go [Go Application] hello.exe
5^done,stack-args=[frame={level="0",args=[]}]
(gdb)
^done
(gdb)
6^done,locals=[]
(gdb)
图1.16 调试Go程序